Trabajo de Fin de Grado

Grado en Ingeniería Informática (GEI)

TALP Online Tracking Tool

Con la colaboración de Barcelona Supercomputing Center (BSC)

24 de Junio del 2019

Autor: Mario Rodríguez Pérez

Directora: Marta García Gasulla

Codirector: Raül Sirvent Pardell

Ponente: Julita Corbalán Gonzalez

Convocatoria: Primavera 2019

Facultat d'Informàtica de Barcelona (FIB)

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Resumen

En aplicaciones HPC en las que se utilice MPI, el reparto de carga entre los procesos es un factor clave que determinará el rendimiento de la aplicación, y por ello ser consciente de los desbalanceos puede ser un comienzo para arreglarlos, en el caso de que sea posible. En este proyecto se ha diseñado e implementado un módulo que es capaz de recoger estas métricas en tiempo de ejecución. Este módulo esta integrado en la librería Dynamic Load Balancing, que al ejecutarse con las aplicaciones modifica su comportamiento para intentar reducir el tiempo de ejecución. Toda la información que se recoja durante la ejecución será puesta a disposición de terceros mediante una API, hecho que pone a esta implementación en ventaja respecto a otras herramientas que desempeñan una funcionalidad similar.

Además, el módulo implementado ha sido validado con dos casos de prueba para comprobar su correcto funcionamiento.También se ha efectuado un análisis de la penalización que provoca el módulo durante la ejecución, y posteriormente se ha analizado el comportamiento de una aplicación para comprobar si los resultados son fidedignos.

Resum

A les aplicacions HPC on s'utilitza MPI, el com es reparteixi la feina és un factor clau. És per això que, ser conscient d'aquests desbalancejos pot ser un bon punt de partida per arreglar-los, en el cas que sigui possible. En aquest projecte s'ha implementat un mòdul capaç de recollir, en temps d'execució, les mètriques relacionades amb MPI, i que es desenvoluparà en el marc de la llibreria Dynamic Load Balancing. Aquesta llibreria quan s'executa amb una aplicació, modifica el seu comportament per intentar reduir el seu temps d'execució. Tota la informació que es generi amb l'execució de les aplicacions amb el mòdul es posarà a disposició de tercers mitjançant una API, fet que deixa a aquesta implementació en avantatge respecte a altres eines que fan una funcionalitat similar.

Abstract

HPC applications where MPI is used, sometimes are not well balanced. This imbalance can be caused, for example, by the distribution of the work in the MPI model. If it's not well done, it can cause long waiting times of processes in MPI directives. For this reason, getting the processes balanced with the others is a priority to get an excellent performance.

The first step to solve this is to know what's happening, and this is what the project wants to solve. Getting the MPI related metrics during execution time and processing them, allow third parties to get all the information. This information can be used to take decisions during execution to improve the performance. This project has been done inside the Dynamic Load Balancing library, which tries to mitigate the unbalance of the MPI model. The implemented module will obtain many metrics of this programming model and will store them in the library structures. Moreover, it will let external agents to read the data to get more information about the execution. Capturing these metrics at execution time will put this tool in competitive advance of other tracing tools that only allow accessing this data once the process has finished.

Agradecimientos

Me gustaría abrir estos agradecimientos nombrando en primer lugar a las personas encargadas del proyecto, Marta y Raul, que me asesoraron tanto en el planteamiento del proyecto como en su desarrollo. También me gustaría agradecerles todos el progreso personal y técnico que he experimentado desde que empecé a trabajar con ellos, en el BSC.
También quiero dar las gracias a Victor, tanto por aguantar las preguntas que a menudo le planteaba, como por ayudarme en la implementación del proyecto.

A Judit, agradecer el apoyo brindado durante estos años de carrera y en especial durante estos largos meses del proyecto, por su paciencia infinita y apoyo incondicional.
A Luz, mi madre, que gracias a sus habilidades gramaticales ha podido cubrir, de cierta manera, mis carencias en el noble arte de la escritura en la legua castellana.

A Oleksandr, Marc y Aleix por la diversión y los impagables desayunos que nos permitieron desconectar durante un breve lapsus de tiempo y retomar el trabajo con energías renovadas.

Por último, a quien también forma parte del grupo anterior, Joel que durante estos 4 años de facultad me ha aguantado, como compañero y amigo, en muchas de las asignaturas ya superadas.

Índice general

Índice de figuras

Índice de cuadros

Índice de Código

Acrónimos

API Application Programming Interface. 13, 21, 22, 25, 30, 34, 46

DLB Dynamic Load Balancing. 9, 13, 17, 19, 23, 25, 29–33, 35–39, 41–43, 46, 47, 49, 51, 53–55, 61, 62, 66, 70, 81, 83

HPC High-Performance Computing. 1, 2, 9, 13, 17, 23, 29, 42, 46, 83

MPI Message Passing Interface. 2, 3, 5, 6, 9, 13, 18, 25, 31, 33, 34, 37–39, 45, 47–49, 52, 53, 55, 56, 58, 60–62

Capítulo 1

Introducción

Este proyecto, que es un Trabajo de Fin de Grado que se incluye en la parte obligatoria del Grado en Ingeniería Informática de la Universidad Politécnica de Catalunya, se ha desarrollado en colaboración con el Barcelona Supercomputing Center - Centro Nacional de Computación (BSC-CNS) y con la finalidad de extender la librería llamada DLB [10].

DLB es una librería que se utiliza durante la ejecución de aplicaciones híbridas. Las aplicaciones híbridas son aquellas que utilizan diferentes modelos de programación paralela, como el modelo de memoria compartida o el modelo de paso de mensajes. Esta librería gestiona los recursos computacionales, aprovechando los recursos que no están siendo utilizados, con la finalidad de conseguir una reducción en el tiempo de ejecución.

1.1. Motivación

El proyecto tiene tres motivaciones principales, que son:

- Dar mas información al usuario sobre el comportamiento de las aplicaciones en el ámbito de HPC durante su ejecución. A pesar de que esta funcionalidad no es nueva, el usuario no tendrá que instrumentar el código para utilizarla. Y al acabar la ejecución, el módulo hará un breve reporte sobre las métricas recogidas en la misma.

- Dar información a los gestores de tareas [I] sobre la utilización de los recursos, con lo que , el gestor podrá implementar nuevas políticas de planificación, adecuándose a esa utilización.

- Obtener métricas para que estén disponibles en las estructuras internas de DLB. Con ellas se podrán programar diferentes políticas de planificación de DLB para poder optimizar más la utilización de recursos en las aplicaciones.

[I] Son los programas encargados de ejecutar de manera aislada las tareas que le envían los usuarios

1.2. Estado del arte

Actualmente hay muchas herramientas que son capaces de coger métricas de las aplicaciones y un gran surtido de ellas pueden ser utilizadas para que los usuarios sean capaces de observar qué está pasando en sus aplicaciones. Muchas herramientas de HPC ofrecen este servicio, de manera que después de la finalización del programa, y a través de una interfaz, el usuario puede conocer estas estadísticas. Como antes se ha matizado, estas aplicaciones permiten la visualización al acabar la ejecución, es decir, que no existe la posibilidad de poder saber en tiempo de ejecución si nuestra aplicación está desaprovechando recursos o si va bien.

1.2.1. Extrae

Extrae[5] es una herramienta desarrollada por el BSC y se utiliza para tracear un aplicación que se ejecute con modelos de programación de memoria compartida o de paso de mensajes(MPI). Una vez compilado un programa con instrumentación y ejecutado junto con Extrae, el programa generará una traza con toda la información relativa a la ejecución. Esta traza podrá ser visualizada posteriormente en Paraver[15]. Aunque Extrae puede conseguir muchas estadísticas sobre la ejecución del programa, no es capaz de suplirlas durante la ejecución, dado que es una herramienta de traceo. Es decir, se guarda información temporal sobre algunos de los eventos que tienen lugar durante la ejecución, pero se deja la interpretación de la misma a otra aplicación.

1.2.2. Intel MPI Tracing

Funcionalidad de MPI tracing de Intel. Provee estadísticas a postmortem, es decir cuando el proceso ya ha acabado, como el tiempo que se pasa en las llamadas o número de veces que se ha llamado a la librería. Esta funcionalidad varía con cada versión de MPI [13] y toda la información que recaba puede ser utilizada únicamente después de la ejecución.

1.2.3. Vampir

Herramienta desarrollada por *Center for Applied Mathematics of Research, Center Jülich*[14] que funciona de una manera similar a Extrae dado que también es un tracer. Colecta eventos que pueden ser visualizados, después de la ejecución, y posteriormente analizados.

1.2.4. MPI tool Interface: Mvapich and TAU

Este módulo de TAU (Tuning and Analysis Utilities)[17], es capaz de recoger estadísticas en tiempo de ejecución y aplicar políticas para mejorar el rendimiento en las aplicaciones. Utiliza la interfaz ofrecida por MPI, MPI_t, para funcionar, hecho que obliga a utilizar una implementación de MPI que aplique el estandard 3.0 o posterior. Además, cada implementación de MPI difiere en la especificación de la interfaz, cosa que complica su integración con todas las implementaciones.

Actualmente el módulo de TAU es compatible con mvapich[II] hecho que también limita las aplicaciones en las que se puede utilizar.

[II] Es una implementación de MPI

1.3. Actores Implicados

En referencia a los actores implicados, se pueden distinguir dos grupos, un primer grupo implicado en el desarrollo del proyecto y un segundo grupo formado por los beneficiarios o potenciales beneficiarios del mismo.

- **Desarrollador:** será la persona encargada del desarrollo del proyecto, que también se hará cargo de las fases de diseño, implementación, validación y evaluación.
- **Director y codirector:** serán los responsables indirectos del proyecto y verificarán el trabajo realizado y quienes harán el seguimiento del proyecto. Realizarán reuniones periódicas con el desarrollador, normalmente cada dos semanas, para comprobar que el proyecto avanza según lo previsto.
- **Ponente:** será la persona encargada de verificar que el proyecto se está realizando según la normativa UPC.
- **Empresa:** será el Barcelona Supercomputing Center, que otorgará todos los recursos para que el proyecto se lleve a cabo. Estos recursos abarcarán desde el soporte en el cual el desarrollador programará, hasta las horas de computación en diversas máquinas con el fin de validar y evaluar el proyecto.
- **Beneficiarios:** cabe distinguir tres tipos de beneficiarios. En primer lugar, los usuarios, que podrán observar el comportamiento de su código durante la ejecución. En segundo lugar, el equipo de DLB, que después de este proyecto, podrá diseñar otros módulos que utilicen estas métricas para tomar decisiones en tiempo de ejecución. En última instancia se beneficiarán los programadores de los diferentes gestores de colas[III] que a través de una API podrán consultar estas estadísticas en tiempo de ejecución.

[III] Programas encargados de gestionar las tareas enviadas por los usuarios y ejecutarlas según las condiciones especificadas

Capítulo 2

Contexto

En los últimos años, con el estancamiento de la conocida Ley de Moore [20] , se ha buscado aumentar el rendimiento de los programas a base de incrementar el número de procesadores por chip así como el número de nodos que conforman un clúster. Este cambio de rumbo ha provocado que las aplicaciones que intentan explotar estas CPUs añadidas, ejecutando su código de manera paralela, tengan dificultades para gestionar estos recursos. Este cambio de paradigma exige superar nuevos retos para conseguir que las aplicaciones hagan un uso más eficiente de los recursos.

2.1. HPC

Actualmente, en la investigación científica, se necesitan hacer simulaciones de todo aquello que es costoso de reproducir en la realidad. El único problema es que estas simulaciones son computacionalmente complejas, y para ello se necesita o mucho tiempo o mucha potencia de cálculo.

Al necesitar tanta potencia de cálculo, surge el denominado High Performance Computing, que consiste en la agrupación de máquinas y coordinación de las mismas para conseguir esa potencia. Con este cambio de paradigma surgen nuevos problemas asociados y en consecuencia nuevas lineas de investigación para solucionarlos.

2.2. OpenMP

OpenMP [3] se define como una API para la ejecución paralela mediante memoria compartida. Se basa en directivas en el código de las aplicaciones, mediante las cuales, el usuario indica qué comportamiento quiere obtener en esa ejecución paralela. Actualmente, casi todos los compiladores más populares soportan este modelo de programación, que es muy utilizado.

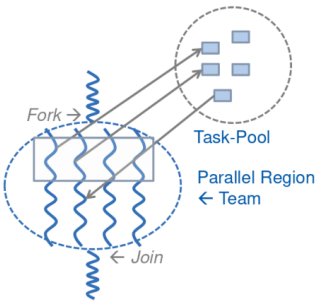

Figura 2.1: OpenMP fork and join model

Esta API utiliza el modelo de ejecución *fork and join*, que cuando el programa se encuentra una región paralela hará un *fork*, que creará los flujos de procesamiento, y al finalizar hará el *join* destruyendo los flujos anteriores.

2.3. OmpSs

El modelo de programación OmpSs, que pertenece al BSC-CNS, fue desarrollado por el departamento de *Computer Science - CS* [4], más en concreto por el equipo de *Programing Models - PM*.

Este modelo de programación para ejecuciones paralelas se basa en el modelo de memoria compartida, y tiene como objetivo extender las funcionalidades de *OpenMP* para hacerlo más heterogéneo y soportar paralelismo asíncrono. Va asociado a dos herramientas que hacen posible su compilación y su ejecución, que son Mercurium, el compilador, y Nanos++ que es el runtime; ambas forman parte del BSC. Al igual que OpenMP, este modelo se sustenta en el hecho que todos los threads comparten el mismo espacio de direcciones.

OmpSs se implementa con el uso de unas directivas de alto nivel y soporta arquitecturas heterogéneas, de manera que en la ejecución de una aplicación puedan ser utilizados más de un tipo de procesador dando lugar a escenarios donde las GPUs y las CPUs puedan ser utilizadas al mismo tiempo. Al dar soporte a directivas de OpenCL y CUDA, este modelo permite que haya partes del código que se ejecuten de una manera más rápida en otros dispositivos, para disminuir así el tiempo de ejecución. [16]

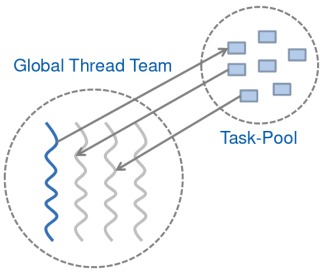

Figura 2.2: Fork Join versus Thread Pool [18]

2.4. MPI

MPI, conocido como Message Passing Interface, es un protocolo de comunicación entre procesos. Es extensamente utilizado en HPC, ya que pasa a ser necesario cuando las aplicaciones se ejecutan en varios nodos y en ese caso el modelo de programación de memoria compartida no podrá utilizarse para la comunicación.

Esta interfaz crea, junto a la aplicación, una topología asociada, en la cual los procesos se pueden pasar mensajes entre ellos. Por ejemplo, los procesos pueden utilizar comunicadores punto a punto para pasar información a procesos específicos.

2.5. DLB: Dynamic Load Balancing

Dynamic Load Balancing, pertenece al BSC-CNS, en concreto al departamento de *Computer Science - CS* y es la tesis doctoral de la doctora Marta García Gasulla. Esta librería fue implementada por la Dra. Garcia y por por el Sr. López.

La librería se utiliza en entornos de High Performance Computing con el fin de intentar aprovechar al máximo los recursos disponibles en las máquinas. Está diseñada para trabajar en entornos paralelos en los que se utilicen aplicaciones híbridas [6], que a menudo tienen un desbalanceo [I] importante en su código. Este desbalanceo puede ser provocado por los modelos de programación utilizados, y como consecuencia los recursos asignados para su ejecución quedan parcialmente desaprovechados, problema que podría ser parcialmente solucionado utilizando DLB y sus políticas para la optimización en el uso de los recursos.

Además, DLB es completamente transparente al programador y si un usuario quiere utilizarla, no se verá obligado a modificar ninguna parte de su código, únicamente deberá cargar DLB dinámicamente en el inicio de la aplicación, que se entrelazará con las diferentes capas de software presentes en la ejecución. Es más, DLB interceptará las llamadas necesarias para aplicar sus optimizaciones, pero se debe tener en cuenta que no será estrictamente necesario haber hecho un análisis de rendimiento antes de aplicar la librería.

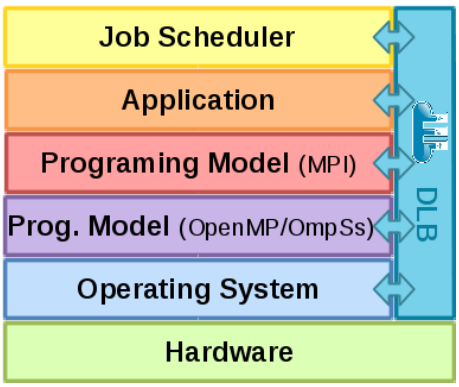

Figura 2.3: Interacción de DLB con el software HPC [10]

[I] Reparto descompensado de la carga de trabajo entre procesos o threads

Capítulo 3

Alcance del proyecto

En esta sección se explicará qué requerimientos son necesarios para desarrollar este proyecto y cuáles son los objetivos que se establecerán como metas del mismo. También se aportarán soluciones a diferentes problemas que podrían aparecer durante su desarrollo y la metodología que se seguirá para desarrollarlo.

3.1. Requerimientos

Los requerimientos para el proyecto son:

- Programar conforme a los criterios de programación de DLB, mantener la independencia de los módulos, programar de manera clara y con buen formato.
- Programar de manera eficiente, es decir no añadir overheadf[I] en las aplicaciones.
- Que las métricas sean fidedignas.
- Que el mecanismo sea transparente a la aplicación.
- Integración con los modelos de programación comúnmente utilizados.

3.2. Objetivos

Los objetivos del proyecto son los siguientes:

- Recoger estadísticas MPI y guardarlas dentro de las estructuras de DLB.
- Implementar una API para la consulta de las estadísticas desde fuera de DLB.
- Validar estas estadísticas y dejarlas listas para su uso.

[I] Tiempo añadido en la ejecución

3.3. Riesgos y posibles soluciones

A continuación se expondrán algunas situaciones de riesgo que pueden darse durante el periodo de desarrollo del proyecto, así como posible solución.

Error en la estimación de tiempo

No es difícil que, en los proyectos, alguna de las fases se retrase debido a la aparición de diversos problemas, tales como: enfermedades, bajas, etc..

Solución: Como este hecho es frecuente, en la planificación inicial se debe dejar un cierto margen de tiempo en cada una de las tareas.

Caída de las colas de Marenostrum 4

Marenostrum 4 se basa en un sistema de colas que permite ejecutar las aplicaciones de manera aislada a las de otros usuarios, y que puede ser suspendido debido a un sobrecalentamiento de las máquinas; esta suspensión puede durar varias horas.

Solución: Contar con márgenes suficientes como para que estos casos no supongan una situación crítica.

Fallo en el equipo

Los equipos tienden a fallar, situación que podría ser critica para el proyecto y su desarrollo.

Solución: Contactar con el support del BSC y esperar a que vuelvan a suministrar el equipo u otro para remplazar el anterior.

3.4. Metodología

En este proyecto se involucrarán cuatro personas, el desarrollador que será el que programará y los demás que revisaran su trabajo. Estas personas han sido especificadas en el apartado 1.3

Para el proyecto se utilizará la metodología SCRUM [21]. Se aplicará esta metodología porque permitirá al equipo del proyecto reaccionar rápidamente ante cualquier problema que surja. Se basa en un sistema de iteraciones que permite adaptarse a los cambios en el diseño, y reaccionar rápidamente ante problemas en el desarrollo. Si el desarrollador encuentra algún impedimento fuera de la reunión periódica, preguntará a los demás integrantes con el fin de resolver la duda y poder proseguir con el desarrollo.

3.4.1. Herramientas

Para el proyecto se utilizarán las siguientes herramientas:

- Trello: se utilizará para el hacer el seguimiento de objetivos y clarificar las tareas de cada período.[19]
- Github: se utilizará para el control de versiones del proyecto. [11]

- Jabber: se utilizará para facilitar la comunicación espontánea, se hará por este servicio de mensajería online o de manera presencial.[12]

Capítulo 4

Diseño e Implementación

En las primeras secciones de este capítulo se explicará, sin profundizar, cómo DLB se acopla en las aplicaciones HPC y qué técnicas usa para mejorar sus tiempos de ejecución. Posteriormente, se explicará qué decisiones de diseño se han sido tomado y cuál es el estado final de la implementación.

4.1. DLB

Actualmente, Dynamic Load Balancing soporta modelos de programación que permitan gestionar de manera dinámica el numero de CPUs en tiempo de ejecución.

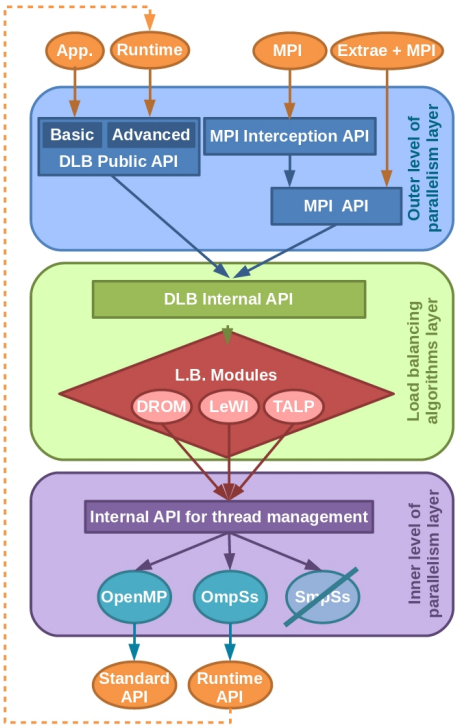

Figura 4.1: Esquema del diseño de DLB [9]

En la figura

Las aplicaciones que utilizan DLB, normalmente se ejecutan con el runtime de OmpSs, llamado Nanos++[1], porque éste, además de poder redimensionar recursos, proporciona información sobre su ejecución, facilita la toma de decisiones en las diferentes políticas de planificación utilizadas por la librería.

La librería se divide en diversos módulos y cada uno de ellos tiene una función especifica e independiente. Tal y como se observa en la figura 4.2, la librería está formada por los siguientes módulos:

- **Kernel**: este es el punto de entrada a la librería cuando un programa se ejecuta con ella. Éste se encarga de reservar todas las estructuras necesarias para el kernel y es el que llama a la inicialización de los otros módulos. Además los coordina y en consecuencia todas las acciones de la librería pasan por él.

- **Lend when Idle o LeWI**: es el encargado de la gestión de las CPUs de cada proceso, y es ejecutado cada vez que la aplicación entra en una llamada MPI. Cuando la aplicación se ejecuta con MPI y LeWI, los procesos dentro de un nodo se coordinan para gestionar las CPUs disponibles en tiempo de ejecución. Cuando un proceso tenga excedentes de CPUs, porque no los está usando para hacer cálculo útil y esté esperando en una llamada MPI bloqueante, cederá sus CPUs a otro proceso. Todas las acciones que lleva a cabo este módulo se pueden conseguir instrumentando el código con su API.

- **Dynamic Resouce Ownership Management o DROM**: el módulo ofrece una API desde la cual un tercero puede activar su remodelación de recursos en tiempo de ejecución, y se encarga de la gestión de los recursos de la aplicación y permite hacer cambios en las máscaras de CPUs de los procesos, permitiendo que se modifiquen durante la ejecución. También puede puede ser utilizado por los gestores de colas, como SLURM [22], para realizar un ajuste en los recursos. Con ello, los gestores pueden liberar recursos previamente reservados si las aplicaciones que se están ejecutando no los utilizan de forma eficiente.

- **Shared Memory**: aunque no se considera un módulo, es el elemento indispensable para el funcionamiento de DLB. Ya que esta Shared Memory es la encargada de la comunicación entre todos los procesos y por ello es indispensable, sin esta memoria, los procesos MPI no podrían verse entre ellos y por consiguiente pensarían que están ejecutándose ellos solos. Pero, sobre todo es indispensable como elemento de comunicación entre los módulos de DLB.

4.1.1. Shared Memory

La Shared Memory es el sistema que utiliza DLB para la comunicación entre procesos MPI de un mismo nodo, y está basado en los Shared Memory Objects de Unix. A pesar de ser compartidos, no todos los procesos tienen que ver necesariamente los mismos objetos. Estas estructuras no pueden ser accedidas directamente desde otros puntos de la librería, pero hay funciones especializadas para hacer lecturas y escrituras en las memorias. La Shared Memory está dividida en dos grandes submemorias:

- **Shared Process Memory**: estructura que contiene información relativa al proceso, tanto máscaras iniciales del mismo, como otras estructuras necesarias para poder efectuar las acciones de las políticas de la librería.

- **Shared CPU Memory**: estructura que alberga toda la información referida a cada CPU, por ejemplo, su estado actual o las estadísticas sobre los tiempos que ha permanecido en cada uno de los estados.

En referencia a las memorias, un thread solo puede acceder a la Shared Memory de todos los procesos MPI del mismo nodo. También cabe destacar que todos el proceso MPI comparte la Shared Memory, es decir que todos los módulos pueden acceder a las mismas estructuras.

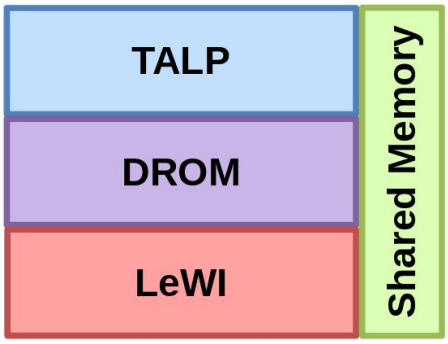

Figura 4.2: Esquema de DLB [9]

4.1.2. MPI Interception

Una de las características más importantes de DLB es el procedimiento que utiliza para interceptar las llamadas de MPI producidas por la aplicación. Este mecanismo se basa en la interfaz de profiling PMPI de MPI, que forma parte del estándar y en consecuencia puede ser utilizada en cualquier implementación de MPI. Esta característica permite al usuario un uso transparente de DLB sin tener que modificar la aplicación, proceso que se puede aplicar en dos momentos:

1. En la fase de enlace [2] de la compilación utilizando librerías dinámicas. Si se enlaza con DLB después de haberse enlazado con MPI, las funciones de DLB remplazarán a las de MPI, interponiéndose.

2. Antes de la ejecución, asignando la variable de entorno LD_PRELOAD y con este método las funciones de DLB remplazarán a las de MPI.

En el caso que una aplicación se enlace con la librería de MPI, tendrá un comportamiento como el que se describe en la imagen 4.3.

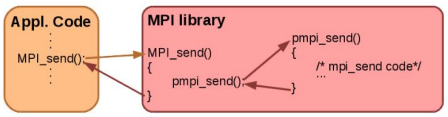

Figura 4.3: Aplicación sin la interpretación de DLB [9]

Si una aplicación utiliza alguno de los dos métodos descritos anteriormente, al llamar a las funciones de MPI, primero se ejecutará el código de DLB y posteriormente el código de MPI. Este hecho permite que se utilicen estos momentos para ejecutar las políticas de DLB. Se observa como la llamada MPI_send DLB hace las veces de wrapper o stub[I].

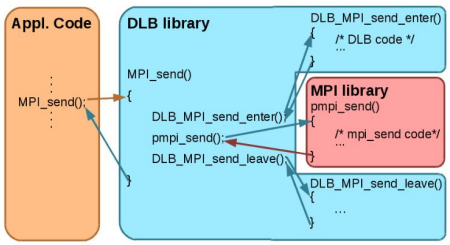

Figura 4.4: Aplicación con la interpretación de DLB [9]

Si hubiera instrumentación (Extrae), el comportamiento sería el descrito en la imagen 4.5, primero se llamaría a Extrae y posteriormente a DLB.

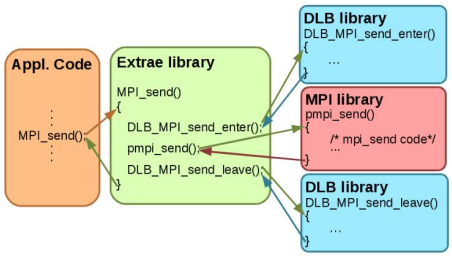

Figura 4.5: Aplicación con la interpretación de DLB y Extrae [9]

Como se observa en la figura 4.5, las rutinas de DLB y Extrae hacen de wrapper a las rutinas más internas. Y teniendo en cuenta que hacen de wrappers, la librería conserva el contexto que tenía al llamar a la capa más interna.

4.1.3. LeWI: Lend When Idle

LeWI [7] [8] es una de las principales políticas aplicadas por DLB y significa Lend When Idle, y se basa en la idea siguiente: cuando una aplicación, que tiene varios procesos, tiene desbalanceo implica necesariamente que unos deben esperan a otros, y en consecuencia esta espera es un uso ineficiente de recursos, dado que estas CPUs no están computando nada.

Para solucionar este problema, DLB presta las CPUs del proceso que espera a otro proceso que aun no haya acabado de computar. Con este mecanismo, los procesos que tienen más carga de trabajo acabarán antes, dado que tendrán mas CPUs para hacer sus cálculos, mientras que los que tardan poco seguirán esperando, pero sin malgastar sus recursos.

[I] Rutina destinada a llamar a otra función

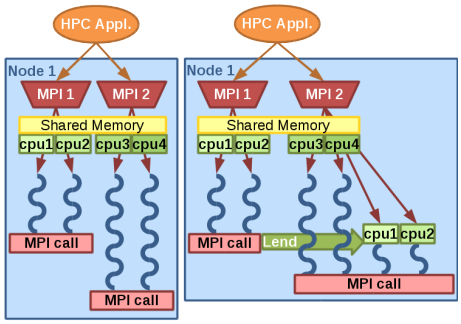

Figura 4.6: Ejemplo de LeWI [9]

En la figura 4.6 se pueden observar los flujos de ejecución de una aplicación que cuenta con dos procesos MPI. También se observa que cada proceso cuenta con 2 CPUs en las que se ejecutan 2 threads. La aplicación está desbalanceada y por tanto un proceso entra antes que el otro a una llamada de espera de MPI. En el caso de ejecutar con DLB, el proceso que ha acabado su parte de cómputo, y ha entrado en la región bloqueante, dejará su CPU al otro proceso (LEND) y el otro proceso aumentará el número de CPUs, lo que le permitirá acabar antes su parte de trabajo.

4.2. TALP

A partir de esta sección se profundizará sobre la parte más técnica del proyecto y se explicará de manera guiada cómo se ha llegado a la finalización del mismo. Se explicarán cuáles han sido las etapas del desarrollo y cómo se ha llegado al razonamiento final pasando por todas las etapas de manera secuencial. Para aclarar como ha implementado módulo, llamado TALP, se adjunta una imagen explicativa en la cual se puede aprecia cómo se relaciona el módulo con la implementación actual.

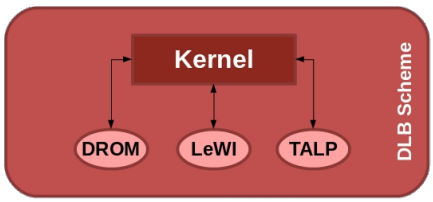

Figura 4.7: Esquema de los módulos DLB

Como se aprecia en la Figura 4.7, todos los módulos de la librería son independientes y solo interactúan con el Kernel. Si un módulo necesita alguna información de otro, llamará a una función del Core para que le proporcione estos datos, comunicación que se efectuará a través de la Shared Memory .

4.3. Métricas de tiempo

Las métricas de tiempo que se pretenden conseguir en este proyecto son aquellas que nos puedan dar información sobre el comportamiento de un programa durante su ejecución. El tiempo que pasan los procesos dentro de llamadas MPI, sea esperando o ejecutándolas , supone un gasto que no aporta nada al programa. Por otra parte el tiempo

de cómputo de los procesos es otra métrica relevante, ya que permite saber la cantidad de trabajo realizado. Gracias a estas dos métricas se puede saber la carga de trabajo que tienen los procesos y en consecuencia, comparando la cantidad de trabajo de todos se puede llegar a saber si el programa esta desbalanceado. Para tener un conocimiento más estricto sobre el tiempo de MPI y el de cómputo de los procesos se establecerá que:

- **Tiempo de MPI**: es la suma de todo el tiempo que pasan las threads en llamadas MPI. Este tiempo se considera inútil desde el punto de vista de la aplicación ya que se gasta en sincronizaciones y paso de mensajes, y no en cálculo.

- **Tiempo de Cómputo**: es el tiempo que pasa una aplicación fuera de MPI. A pesar de llamarse de cómputo no tiene porque dedicarse estrictamente al cálculo, dado que no podemos detectar cuándo un thread está leyendo de memoria o cuándo el gestor de threads está utilizando uno de ellos para sincronización.

Todas estás métricas podrán ser calculadas tanto para una zona delimitada como para la ejecución completa. Teniendo en cuenta que, si queremos obtener las métricas para una zona delimitada se necesitará el uso de una API.

4.4. Gestión de CPUs y tiempo

Para la implementación del módulo se utilizará una librería que permitirá saber qué CPUs están activas en cada proceso, y otra que proporcione información sobre el tiempo transcurrido durante la ejecución.

4.4.1. Gestión de CPUs

El módulo tendrá una parte fundamental que será la encargada de hacer todos los cálculos de las métricas.También deberá tener un control sobre el número actual de CPUs que están activas, con el fin de obtener el tiempo de cómputo del programa. Para mantener ese control se utilizará la libreria **<sched.h>**, que proporcionará máscaras, es decir estructuras que para guardar información relativa a las CPUs. Dichas estructuras, servirán para que el módulo pueda controlar qué CPUs están en MPI y cuáles están computando. Esta librería nos proporciona la struct *cpu_set_t* y las funciones auxiliares para operar sobre las máscaras, tales como:

```
1    void CPU_ZERO(cpu_set_t *set);
2    void CPU_SET(int cpu, cpu_set_t *set);
3    void CPU_CLR(int cpu, cpu_set_t *set);
4    int  CPU_ISSET(int cpu, cpu_set_t *set);
5    int  CPU_COUNT(cpu_set_t *set);
```

Código 4.1: Funciones de CPU Mask

4.4.2. Gestión de tiempo

En referencia a la obtención de las métricas, se ha utilizado una interfaz ya implementada que forma parte de la librería. Esta interfaz utiliza dos structs básicas, basadas en dos librerías de sistema diferentes, y que se utilizan para guardar las siguientes medidas de tiempo:

- **<time.h>**: librería proporciona la struct *timespec*, que puede llegar a contar hasta nanosegundos y tiene un uso muy sencillo, se basa en la llamada *clock_gettime*(*reloj, struct*) y a diferencia de otras librerías que calculan tiempo, necesita dos puntos de tiempo para realizar el cómputo.

- **<sys/time.h>**: librería que nos proporciona *timeval*, que cuenta hasta microsegundos y puede ser utilizada para medir tiempos. Sin embargo es útil en la creación de timers o para consultar el tiempo en un momento determinado. Para obtener el tiempo con timeval se puede recurrir a la llamada *gettimeoftheday*(), pero retorna más información de la necesaria.

Como la librería de DLB tiene unas funciones que permiten la utilización de *timespec* y *timeval*, se evaluará cuál de las dos ofrece mejores características para su utilización. Para comprobar cuál tiene menos overhead, se hará una prueba en la cual se ejecutará el siguiente código:

```
1         struct timespec stop;
2         for (; i<N; i++) {
3             clock_gettime( CLOCK_MONOTONIC, &stop);
4         }
```

Código 4.2: Test Case Timespec

```
1         struct timeval tv;
2       int i = 0;
3       for (; i<N; i++) {
4           gettimeofday(&tv,NULL);
5       }
```

Código 4.3: Test Case Timeval

Comparando la ejecución de estos dos casos, se obtiene la gráfica 4.8, en la que se observa que la diferencia no es significativa. A pesar de ello se elige *timespec* ya que otorga más precisión, llegando hasta los nanosegundos.

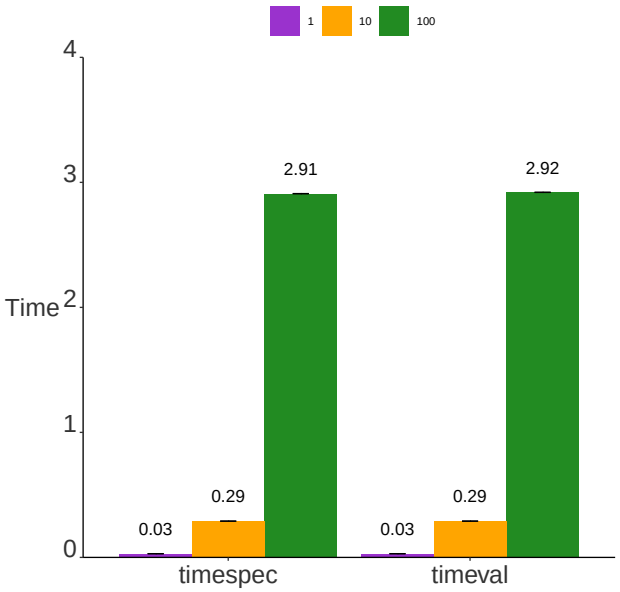

Figura 4.8: Comparación de tiempos entre *gettimeofday* y *clock_gettime*

A continuación se muestra cómo es la interfaz que ofrece DLB para efectuar operaciones de tiempo. Se dispone de operaciones de suma, resta, multiplicación y reset de tiempo así como funciones de conversión a otras structs de tiempo.

```
1  void get_time( struct timespec *t );
2  void get_time_coarse( struct timespec *t );
3  int64_t get_time_in_ns(void);
4  int diff_time( struct timespec init, struct timespec end, struct timespec*
   ↪  diff );
5  void add_time( struct timespec t1, struct timespec t2, struct timespec* sum
   ↪  );
6  void mult_time( struct timespec t1, int factor, struct timespec* prod );
7  void reset( struct timespec *t1 );
8  double to_secs( struct timespec t1 );
9  int64_t to_nsecs( const struct timespec *ts );
10 int64_t timespec_diff( const struct timespec *start, const struct timespec
   ↪  *finish );
11 void add_tv_to_ts( const struct timeval *t1, const struct timeval *t2,
   ↪  struct timespec *res );
12 void ns_to_human( char *buf, size_t size, int64_t ns );
```

Código 4.4: Funciones de la interfaz de tiempo de DLB.

4.5. Implementación del Módulo

Para organizar de la mejor manera posible el proceso de implementación del módulo, se plantaron las siguientes etapas:

- **Declaración de Estructuras**: fase en la que se especificarán las estructuras necesarias para poder recoger las métricas.

- **Obtención de Métricas**: etapa se implementará cómo se recogen estas métricas y se adecuará la especificación para que el módulo trabaje conjuntamente con la librería.
- **Desarrollo de la API**: etapa en la que se integrarán las métricas en las estructuras de la librería y se diseñarán el conjunto de llamadas que habilitará a los usuarios externos para obtener estas métricas.

4.6. Declaración de Estructuras

Al principio se diseñaron las estructuras que se encargarían de almacenar toda la información sobre las métricas. Por ello se dedujo que serían necesarias dos variables para calcular el tiempo transcurrido entre los dos últimos cálculos de métricas, y otras dos para acumular las métricas calculadas. Finalmente todas ellas se guardarán en una nueva estructura llamada *monitor_t* que tendrá la siguiente forma:

```
typedef struct monitor_s {
        struct timespec tmp_mpi_time;       // Temporal MPI Time
        struct timespec tmp_compute_time;   // Temporal Compute Time

        struct timespec mpi_time;           // Total MPI Time
        struct timespec compute_time;       // Total Compute Time

        struct timespec init;           // Elapsed time
        struct timespec end;           // Elapsed time

} monitor_t;
```

Código 4.5: Declaración de monitor_t

Las variables temporales sirven para guardarnos el momento de tiempo en el que se ha hecho la última actualización, *mpi_time* y *compute_time* se utilizan para acumular los valores de tiempo de las diferencias calculadas a partir de los valores *tmp*. Además también se calcula el tiempo transcurrido entre que inicia y acaba MPI con las variables *init* y *end*. Esta estructura se ha implementado para que pueda servir tanto para la ejecución entera como para una región acotada de código 4.8.

Por otra parte y mediante las políticas de DLB, las CPUs pueden estar ejecutando un thread de un proceso o de otro tal y cómo se se explica en la Sección 4.1.3; por ello, para poder calcular correctamente las métricas el módulo también necesita que saber cuántas CPUs hay activas en el proceso. Para ese fin, la estructura encargada de guardar esa información es la siguiente:

```
typedef struct talp_info_s{
     monitor_t  monitoringRegion;

     cpu_set_t active_working_mask;         // Active CPUs
     cpu_set_t in_mpi_mask;                 // CPUs in MPI
     cpu_set_t active_mpi_mask;             // Active CPUs in MPI
} talp_info_t;
```

Código 4.6: Estructura básica de TALP

Si queremos llevar la cuenta de las CPUs de cada proceso, se necesitan como mínimo dos máscaras que controlan cuáles están ejecutando llamadas MPI y cuáles no. Además, también necesita saber si las CPUs estan activas o no dado que la librería permite que un proceso deje alguna de sus CPUs a otros procesos. Todo esto se llevará a cabo utilizando las variables *in_mpi_mask* y *active_mpi_mask*.

En el caso de que un thread entre en MPI y DLB no tenga ninguna política activada, se considerará que todas las CPUs activas entrarán con él.

Toda la estructura llamada *talp_info_t* se integrará dentro del *SubProcess Descriptor* (SPD) de DLB. Dado que el SPD es propio de cada proceso cada proceso tendrá su propia estructura. Inicialmente se había pensado de tal manera que cada una de estas estructuras fuera declarada como variable global, pero en una fase posterior se decidió integrarlo en el SPD, ya que esta estructura que contiene la información propia del proceso como datos sobre la ejecución, la política a utilizar o qué módulos están habilitados. Por lo tanto el SPD queda de la siguiente manera:

```
1  typedef struct SubProcessDescriptor {
2      pid_t
       ↪  id;
3      bool dlb_initialized;
4      bool dlb_preinitialized;
5      bool dlb_enabled;
6      cpu_set_t process_mask;
7      cpu_set_t active_mask;
8      options_t options;
9      pm_interface_t pm;
10     policy_t lb_policy;
11     balance_policy_t lb_funcs;
12     void *lewi_info;
13     //---------------------- Parte añadida -------------------------//
14     bool talp_enabled;
15     bool talp_initialized;
16     void *talp_info;                  // Void pointer to avoid type
       ↪  dependencies.
17 } subprocess_descriptor_t;
```

Código 4.7: Estructura SubProcess Descriptor actualizada

Para saber si se dispone de las estructuras iniciales y si se ha activado el módulo durante la ejecución, se han añadido dos variables llamadas *talp_enabled* y *talp_initialized*, como se puede observar en el Código 4.7.

4.7. Obtención de Métricas

Para obtener las anteriores métricas, DLB deberá interponerse en momentos clave de la ejecución, y realizar llamadas al módulo para que éste pueda calcular métricas. Dichas llamadas se añadirán en los siguientes puntos del Kernel:

- **Llamadas MPI**: se capturará cuándo una CPU entra y sale de MPI.

- **Llamadas de LeWI**: se utilizan para alterar el número de CPUs activas de cada proceso, en el caso de estar la política activa.

4.7.1. Llamadas MPI

DLB intercepta todas las llamadas MPI con el fin de saber en qué momentos puede aplicar las políticas definidas. En la implementación del módulo, se utilizará este punto para saber cuándo se está en MPI y cuándo computando. Más concretamente, se utilizarán algunas de las funciones, que nos facilita el mismo Kernel de DLB, para saber cuándo actuar con el módulo. Además, se añadirán llamadas al módulo en las siguientes funciones:

1. **IntoCommunication**: es ejecutada por los wrappers de MPI 4.1.2 para interponer código al principio de las llamadas.

2. **OutOfComunication**: es llamada al finalizar la función de MPI. En esta función se contabilizará que esta CPU ha entrado a MPI. En el caso de no tener políticas de DLB activas, se entenderá que todas las CPUs han entrado a la vez.

3. **IntoBlockingCall**: se ejecuta al entrar en una llamada bloqueante de MPI y que nos servirá para poder controlar cuándo el kernel devuelva una CPU en el caso de utilizar las políticas de balanceo, como LeWI o LeWI-MPI 4.1.3.

4. **OutOfBlockingCall**: que sirve para saber cuándo acaba una llamada bloqueante de MPI, como *MPI_Barrier* o *MPI_Send*.

4.7.2. Llamadas de LeWI

Por parte de DLB se interceptarán las llamadas que tienen que ver con las políticas definidas anteriormente. El módulo interceptará estos módulos porque son capaces de reajustar la máscara de CPUs de los procesos MPI y ésto hará variar en tiempo el cómputo del proceso. De esta manera se interceptarán las llamadas capaces de modificar el numero de CPUs, y que son las siguientes:

- **Lend**: función utilizada por los procesos que entran a MPI y se quedan bloqueados esperando, con ella dejan las CPUs sobrantes a otros procesos.

- **Acquire**: función utilizada por los procesos que aún están computando. Con ella intentan adquirir más CPUs para acabar antes su parte de trabajo.

En la figura 4.8 se muestra cómo quedará las funciones del Kernel de DLB relacionadas con LeWI, al añadir las funciones del módulo.

```
1 int XXXX_cpu(const subprocess_descriptor_t *spd, int cpuid) {
2     int error;
3
4     if (!spd->dlb_enabled) {
5         error = DLB_ERR_DISBLD;
6     }
7      else {
8         add_event(RUNTIME_EVENT, EVENT_LEND);
9         error = spd->lb_funcs.ZZZZ_cpu(spd, cpuid);
```

```
10          if (error == DLB_SUCCESS && spd->options.talp) {
11              talp_cpu_YYYY(cpuid);
12          }
13          add_event(RUNTIME_EVENT, EVENT_USER);
14      }
15      return error;
16  }
```

Código 4.8: Función modelo de LeWI en la cual se ejecuta TALP

Las dos funciones en las que se añadirá código del módulo serán el DLB_Lend y DLB_Acquire, que son las principales funciones en las que se observa si una CPU se activa o se desactiva.

4.7.3. Implementación

Tal y como queda establecido en la sección 3.1, la premisa básica del módulo implementado es su independencia de los otros que forman parte de DLB; para ello este módulo no debe realizar ninguna llamada a los otros módulos. Sus funciones son las siguientes:

```
1  // Initializes the module structures
2  void talp_init( subprocess_descriptor_t* spd);
3
4  // Finishes the execution of the module and shows the final report if
   ↪  needed
5  void talp_finish( subprocess_descriptor_t *spd);
6
7  // Enables the cpuid for the current process
8  void talp_cpu_enable(int cpuid);
9
10 // Disables the cpuid for the current process
11 void talp_cpu_disable(int cpuid);
12
13 // Update the metrics when entering MPI
14 void talp_in_mpi();
15
16 // Update the metrics when going out MPI
17 void talp_out_mpi();
18
19 // Update the metrics when entering MPI blocking call
20 void talp_in_blocking_call();
21
22 // Update the metrics when going out MPI blocking call
23 void talp_out_blocking_call();
24
25 // Function to print the stats of the process
26 void talp_mpi_report();
27
28 // Initialize necessary structures
29 void dlb_talp_mpi_init();
30
```

```
31 // Handler of the Finalize of MPI
32 void dlb_talp_mpi_finalize();
```

Código 4.9: Funciones implementadas en TALP

Por otra parte, a las funciones de DLB que hacen el reconocimiento de argumentos se habrán tenido que incorporar funciones con el fin de que se pueda habilitar y deshabilitar el módulo. También se han añadido funciones para poder elegir qué tipo de resumen se quiere obtener en la ejecución de la aplicación.

4.8. Desarrollo de la API

También se pondrá a disposición de terceros una interfaz que les permitirá acceder a los datos recabados durante la ejecución.

Esta interfaz permite que un programa externo obtenga las estadísticas de los procesos que se estén ejecutando junto a DLB y TALP. Así, este programa externo será capaz de acoplarse a la Shared Memory 4.1.1 del proceso y de obtener información sobre las métricas que está recogiendo TALP.

```
1  // Attach current process to DLB system as TALP
2  int DLB_TALP_Attach(void);
3
4  // Detach current process from DLB system
5  int DLB_TALP_Detach(void);
6
7  // Get the number of CPUs
8  int DLB_TALP_GetNumCPUs(int *ncpus);
9
10 // Get \acrshort{mpi} Time of the current process
11 int DLB_TALP_MPITimeGet(double* mpi_time);
12
13 // Get Compute Time of the current process
14 int DLB_TALP_CPUTimeGet(double* compute_time);
```

Código 4.10: API para obtener las estadísticas recogidas por TALP

También se han implementado unas funciones con las que cualquier usuario puede acotar una región de código para saber las estadísticas de la misma. Funciones que tienen la forma siguiente:

```
1 // Start (or unpause) monitoring region
2 int DLB_MonitoringRegionStart(dlb_monitor_t handle);
3
4 // Stop (or pause) monitoring region
5 int DLB_MonitoringRegionStop(dlb_monitor_t handle);
6
7 // Print a Report by stdout of the monitoring region
8 int DLB_MonitoringRegionReport(dlb_monitor_t handle);
```

Código 4.11: Cabeceras de MonitoringRegions en C

Además, como en el ambito del HPC se utiliza Fortran en diversas aplicaciones, se decidió elaborar una interfaz para que estas funciones puedan ser llamadas desde Fortran 77 y 90 mediante bindings[II] a funciones de la librería escritas en C. Además también servirá para poder ejecutar aplicaciones escritas en Fortran para el apartado de validación y evaluación posteriores.

```
1         function dlb_monitoringregionregister(region_name)              &
2       &  result (handle) bind(c, name='DLB_MonitoringRegionRegister')
3             use iso_c_binding
4             type(c_ptr) :: handle
5             character(kind=c_char), intent(in) :: region_name(*)
6         end function dlb_monitoringregionregister
7
8         function dlb_monitoringregionstart(handle)                      &
9       &         result (code) bind(c, name='dlb_monitoringregionstart')
10            use iso_c_binding
11            integer(kind=c_int) :: code
12            type(c_ptr) , value, intent(in) :: handle
13        end function dlb_monitoringregionstart
14
15        function dlb_monitoringregionstop(handle)                       &
16      &         result (code) bind(c, name='dlb_monitoringregionstop')
17            use iso_c_binding
18            integer(kind=c_int) :: code
19            type(c_ptr) , value, intent(in) :: handle
20        end function dlb_monitoringregionstop
21
22        function dlb_monitoringregionreport(handle)                     &
23      &         result (code) bind(c, name='DLB_MonitoringRegionReport')
24            use iso_c_binding
25            integer(kind=c_int) :: code
26            type(c_ptr) , value,  intent(in) :: handle
27        end function dlb_monitoringregionreport
```

Código 4.12: Cabeceras de MonitoringRegions en Fortran

En el siguiente fragmento de código se encuentra un caso de uso en el cual un usuario quisiera saber cómo de desbalanceada está su función. Para ello añadiría dicha función con las funciones de DLB, tal y como se muestra en la figura 4.13.

```
1 dlb_monitor_t handle = DLB_MonitoringRegionRegister("My Region");
2 main_loop {
3    ...
4    DLB_MonitoringRegionStart(handle);
5    function_to_be_monitorized();
6    DLB_MonitoringRegionStop(handle);
7    ...
8 }
9 monitor_info_t info;
```

[II] Mapping de las funciones de un lenguaje a otro para que puedan ser utilizadas

Código 4.13: Ejemplo de utilización de las MonitoringRegions

4.8.1. Configuración del TALP

DLB cuenta con unas variables de entorno que pueden ser asignadas, y cuya finalidad es alterar su comportamiento. Además, se han añadido a DLB las directivas necesarias para poder alterar el comportamiento del módulo. Con ellas se puede decidir qué se hará con todos datos al acabar la ejecución. Estas funcionalidades se presentan al usuario de la siguiente manera:

Flag	Default Value	Type
–talp:	no	(bool)
–talp-summary:	none	[app:node:process:iteration:omp]

Cuadro 4.1: Opciones del módulo

Si el usuario quiere activar el módulo deberá añadir *–talp* en el DLB_ARGS, que es una variable de entorno que se utiliza para activar funcionalidades de DLB; por el contrario si no está presente, el módulo no se activará para no añadir overhead.

Además, el usuario podrá especificar qué tipo de resumen quiere al final de la ejecución, pidiendo los siguientes reportes:

- **Node**: Resumen de la ejecución a nivel de nodo. Cada master thread del nodo recogerá todos los datos de su nodo y los escribirá en el stderr.
- **Process**: Cada proceso escribirá sus estadísticas locales en el stderr.

Capítulo 5

Validación

En este capítulo se comprobará que los resultados que reporta el módulo concuerdan con la ejecución del programa, es decir que son fidedignos, como se especifica en los requisitos, sección 3.1.

5.1. Entorno de validación

5.1.1. Máquina

Para la ejecución de esta validación de resultados, se utilizará la máquina Marenostrum 4, que está formada por nodos que tienen 2 Xeon Platinum 8160, 24 cores cada uno, es decir, 48 cores por nodo, basado en la arquitectura *x86_64* . También cabe destacar que la red de interconexión de los mismos se basa en la tecnología Intel Omni-Path que otorga una velocidad máxima de 100Gbit/s. El supercomputador ejecuta la versión 4.4.12 del kernel de Linux y utiliza la versión 17.11.7 de SLURM.

Esta máquina dispone de un sistema de módulos [I], que se basa en la alteración de las variables de entorno del usuario y que permite a los usuarios cargar diferentes versiones de las aplicaciones sin tener que instalar manualmente nada. Este sistema será el que se utilizará para preparar los entornos.

5.1.2. Software

Para la validación se utilizarán IMPI 2017.4 [II], como versión de MPI, y Mercurium, como compilador. Será necesario utilizar Mercurium[III], ya que las aplicaciones se ejecutarán con OmpSs, y es el único compilador que lo soporta. Dado que se utilizará Mercurium y OmpSs en Marenostrum4, se utilizará el módulo llamado *ompss/git* que contiene tanto la última versión de OmpSs como su versión de Mercurium asociada.

Para la compilación de la librería se utilizará la misma versión de MPI que la utilizada en las aplicaciones, IMPI 2017.4, y GCC 7.2.0, ya que es una de las versiones más estables disponibles en la máquina.

[I] https://modules.sourceforge.net/man/module.html
[II] https://software.intel.com/en-us/mpi-library
[III] Compilador que soporta el modelo de programación de OmpSs

Por otro lado, también se utilizará Extrae como librería de instrumentación en su versión 3.5.4, dado que era la versión más reciente en el momento de iniciar el proyecto.

5.1.3. Aplicaciones

Para la validación del proyecto se utilizarán las aplicaciones:

- **Pils**: esta aplicación sintética forma parte de los ejemplos de DLB y sirve para simular de manera artificial un desbalanceo.
- **Alya**: esta aplicación de HPC resuelve diferentes sistemas físicos. Se basa en un sistema de componentes en el cual cada uno hace referencia a un tipo de comportamiento físico diferente.

5.1.4. Método

Para la validación del proyecto se aplicarán dos métodos, asociados a dos aplicaciones:

- **Validación por código fuente**: Este método de validación se utilizará en la aplicación Pils, ya que permite personalizar el desbalanceo que tendrá en tiempo de compilación. Se comparará ese desbalanceo con los tiempos reportados por el módulo a la finalización de la ejecución.
- **Validación por trazas**[IV]: Método que se utilizará en el caso de Alya y que se basará en la obtención de trazas para comprobar el correcto funcionamiento del módulo. Este tipo de validación requerirá que el binario esté instrumentado con el fin de poder ejecutar la aplicación con Extrae y de visualizar las trazas correctamente. Los resultados obtenidos en las trazas se compararán con los tiempos de ejecución de una zona acotada del código. Esta zona utilizará la API que presenta el módulo para delimitar zonas e imprimir los resultados.

En los dos casos anteriores las aplicaciones tendrán que están enlazadas a DLB, ya que será necesario para que DLB pueda iniciarse, y para ello se utilizará el enlace en tiempo de compilación.

[IV] Archivos generados por Extrae

5.2. Pils

5.2.1. Validación por código fuente

En esta sección se ejecutará Pils, que es una aplicación sintética y que será compilada con diferentes grados de desbalanceo y ejecutada con diferentes políticas de DLB. En el programa se puede especificar la cantidad de trabajo que tiene cada proceso, así que se podrán simular diferentes grados de reparto de trabajo, fijando la siguiente variable global en el código fuente.

```
1 const int DEF_LOADS_SIZE = 2;
2 const int default_loads[] = {Load1,Load2};
```

En las ejecuciones se validarán los datos de DLB, generando 3 combinaciones de políticas:

- **TALP**: Se ejecutará únicamente con el módulo implementado.
- **TALP+LeWI**: Se basará en el caso anterior añadiendo la política LeWI.
- **TALP+LeWI+LeWI-MPI**: Se basará en el caso anterior añadiendo la política LeWI-MPI.

Para no aumentar mucho el número de casos se ejecutarán los 3 grados siguientes de desbalanceo del Pils:

- **Load1**=100 **Load2**=900
- **Load1**=450 **Load2**=550
- **Load1**=500 **Load2**=500

Se han elegido estas opciones para simular un caso con una reparto con mucho desbalanceo (100-900), otro con poco desbalanceo (450-550) y un último caso sin desbalanceo (500-500). Las gráficas se mostrarán agrupando las ejecuciones por las combinaciones de políticas que se han utilizado. También se ejecutará la aplicación 10 veces por cada caso, para minimizar resultados extremos, y se calculará la desviación estándar asociada a la medida, la cual se mostrará en la gráfica correspondiente.
Las ejecuciones se efectuarán con 2 procesos MPI, uno por cada cantidad de trabajo, cada uno tendrá 2 CPUs, lo que facilitará ver el comportamiento del módulo en aplicaciones híbridas.
Además se dispone de un Makefile que se utilizará para compilar las diferentes versiones de los ejecutables binarios . Este Makefile permite que se pueda enlazar DLB a dichos ejecutables en tiempo de compilación.

5.2.2. Resultados

Las gráficas siguientes han sido elaboradas con los valores que el módulo escribe al acabar la ejecución.

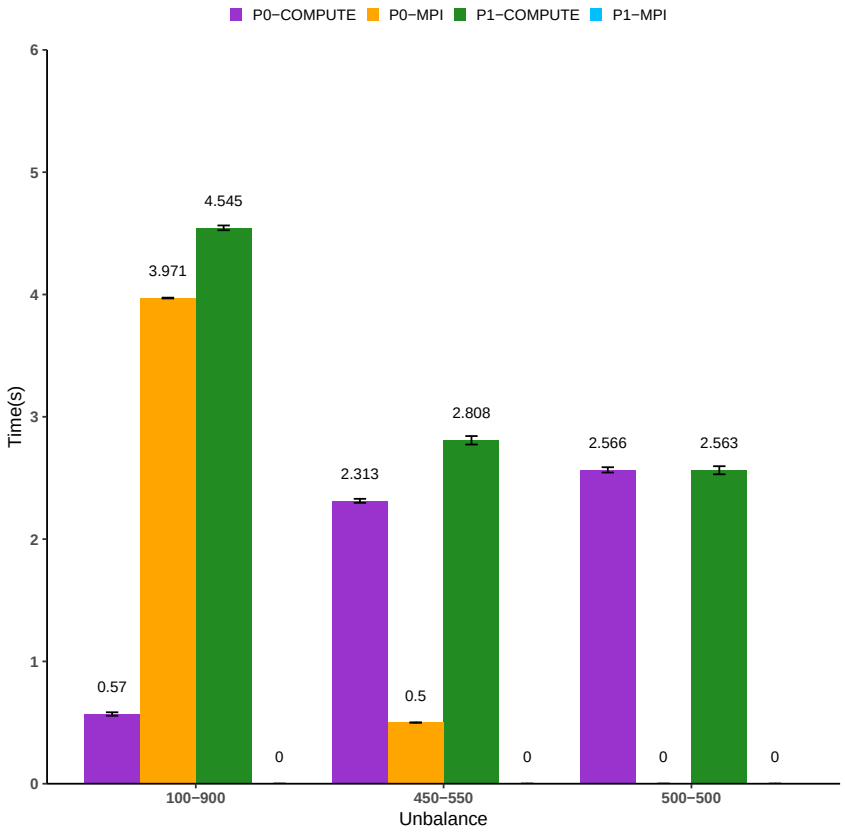

Figura 5.1: Políticas: Caso TALP

En la gráfica 5.1 se describen los tiempos de los dos procesos en los 3 grados de desbalanceo, así como la desviación estándar de las medidas. Se observa cómo en el primer caso el proceso P0, que tiene una carga de 100, pasa la mayor parte del tiempo en llamadas MPI, mientras que el proceso P1 pasa la totalidad del tiempo en cómputo.

Este hecho se aprecia también en los otros casos, siendo totalmente coherente con el desbalanceo descrito en el código de la aplicación.

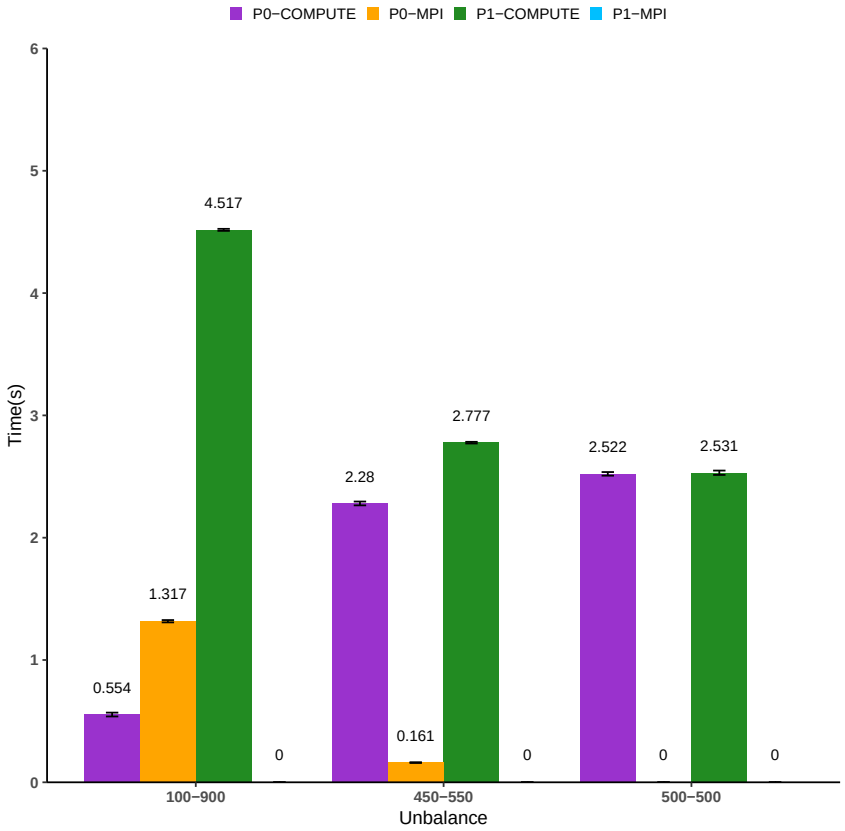

Figura 5.2: Políticas: Caso TALP+LewI

En la gráfica 5.2 se aprecia, para el conjunto de los procesos, una disminución de los tiempos de MPI respecto al caso de la gráfica 5.1; el hecho es debido a la política de DLB LeWI. Se aprecia también como las CPUs del proceso P0 están siendo cedidas al proceso P1 para que acabe antes. El tiempo que se pasan las CPUs en las llamadas MPI disminuye. Si se compara este caso con el anterior, se puede apreciar una mejora del tiempo de ejecución y por tanto el programa acaba antes. Los casos más desbalanceados resultarán beneficiados si se ejecutan con DLB, sin embargo los muy balanceados saldrán un poco perjudicados ya que tanto el TALP y como DLB añadirán overhead a su ejecución.

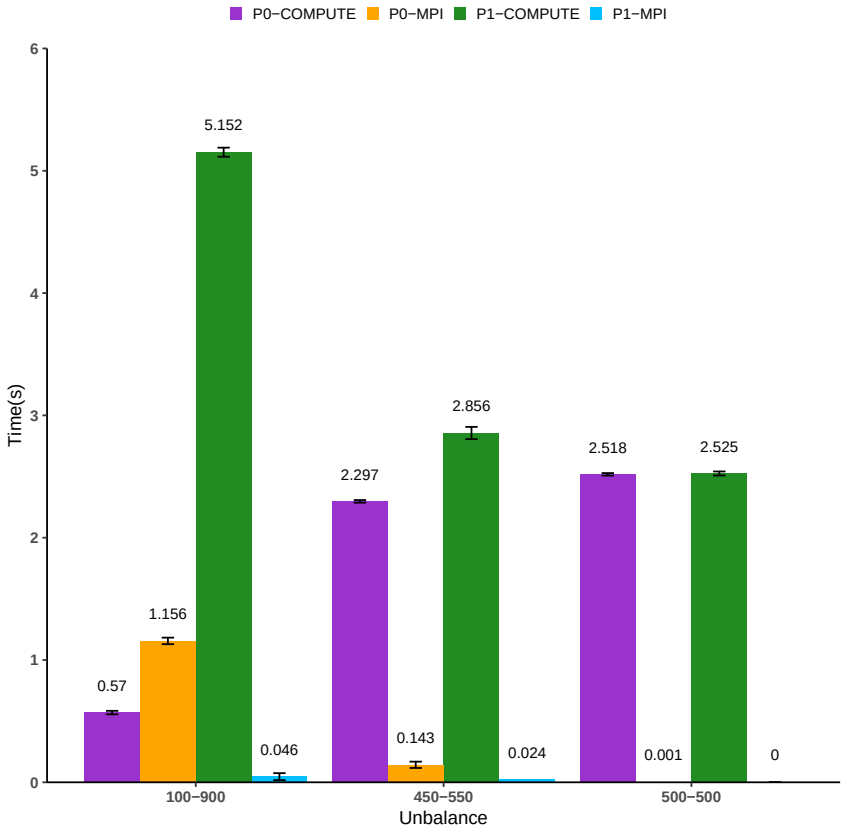

Figura 5.3: Políticas: Caso TALP+LeWI+LeWI-MPI

En la Figura 5.3 se observa cómo se esta invirtiendo más tiempo en computar que en el segundo caso [5.2], es decir, que la política LeWI-MPI no está resultando del todo beneficiosa ya que se está perdiendo rendimiento. En estas ejecuciones se ha aumentado el tiempo de cómputo, a pesar de ser el mismo trabajo.

Si pensamos en la cantidad de trabajo, podríamos deducir que siempre será la misma sin depender de cuantas CPUs haya activas, es decir; que el trabajo que hace una CPU se podrá realizar en la mitad de tiempo con 2 CPUs. Sin embargo, este hecho no es siempre cierto ya que las aplicaciones se ejecutan con OmpSs y el runtime de éste puede influir en la ejecución y puede retener las CPUs hasta que todas acaben el trabajo.

5.2.3. Justificación de los tiempos

Al comparar la gráfica de TALP+LeWI, Figura 5.2, con la de TALP+LeWI+LeWI-MPI, Figura 5.3, se puede observar que el tiempo de cómputo del proceso 0 aumenta, hecho extraño si se considera que las dos ejecuciones tienen el mismo trabajo. Por ello se ha instrumentado la aplicación y procedido a explicarlo mediante trazas. En todas ellas se analizará el comportamiento del proceso P1, que es el que tiene más carga de trabajo.

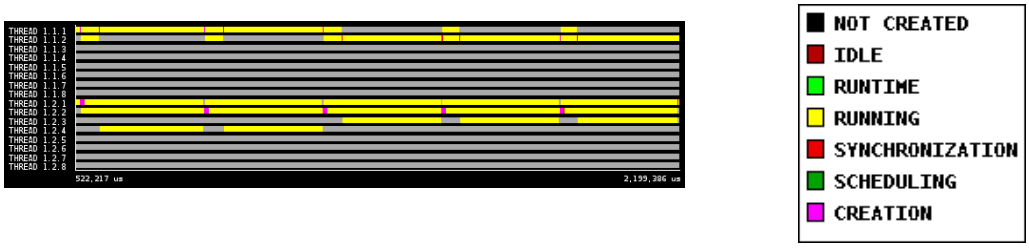

Figura 5.4: Thread State caso TALP+LeWI

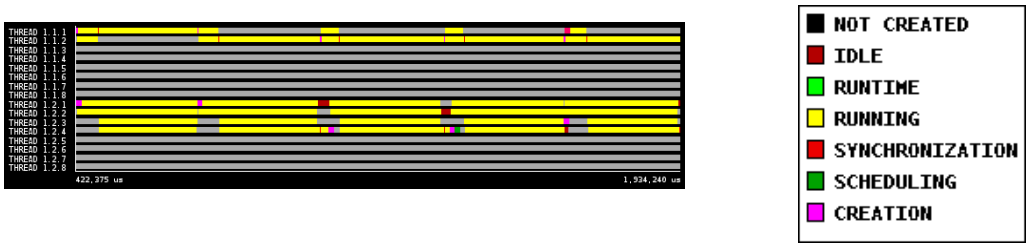

Figura 5.5: Thread State caso TALP+LeWI+LeWI- MPI

En las trazas 5.4 y 5.5 se muestra la actuación de las políticas de DLB, hecho que se constata viendo cómo los procesos aumentan el número de CPUs que tienen activas a un valor superior al que les había sido otorgado (2). Con el fin de de analizar el origen de la diferencia en los tiempos entre las gráficas 5.2 y 5.3, se mostrarán las tablas generadas a partir de los histogramas asociados a las figuras 5.4 y 5.5, y posteriormente se computará la diferencia entre ellas.

	IDLE	RUNTIME	RUNNING	SYNC	SCHEDULING	CREATION	WAKING UP	STOPPED
LeWI	38,106.11	22,722.44	4,632,387.75	1,775.02	18,881.94	13,122.88	129.88	8,665,288.17
LeWI-MPI	117,335.13	13,044.41	5,343,129.44	284.76	34,378.57	15,184.21	74.67	6,563,909.01
Diferencia	79,229.02	-9,678.03	710,741.69	-1,490.26	15,496.63	2,061.33	-55.21	-2,101,379.16

Cuadro 5.1: Tabla de diferencias

Entre todos los estados de CPU representados en la tabla anterior no se debería contar la columna STOPPED dentro del tiempo de cómputo del módulo, ya que los threads no estarán activos. Como se observa en la tercera fila, columna RUNNING de la tabla 5.1, la diferencia de RUNNING entre los casos es de 0.71 segundos, que resulta ser una diferencia significativa. Y dado que la ejecución para obtener trazas se realiza con una versión instrumentada, que provoca overhead, se puede afirmar que la diferencia, con todo y ser grande, es relativa.

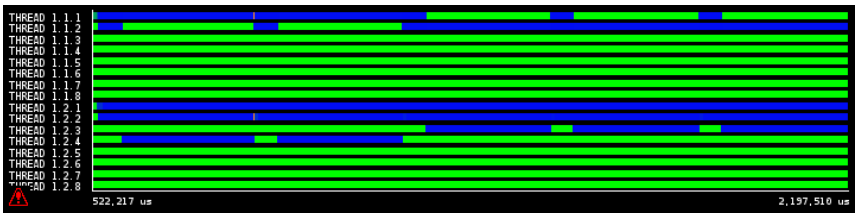

Figura 5.6: Ciclos por microsegundo del caso TALP+LeWI

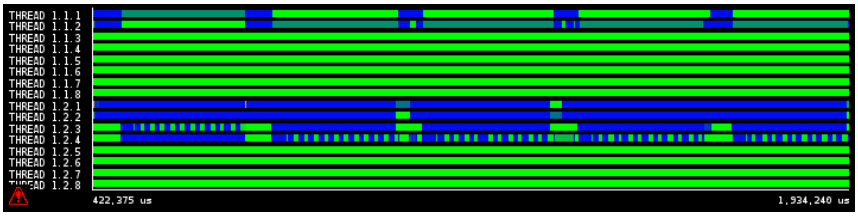

Figura 5.7: Ciclos por microsegundo del caso TALP+LeWI+LeWI-MPI

Gracias a la observación de la traza 5.7 se deduce que el proceso 0 está perdiendo rendimiento en una de las CPUs, ya que los ciclos por segundo oscilan mucho. Comparando la traza del caso 3 con la del caso 2, se observa que unicamente en el caso 3 existen regiones en las que los ciclos por microsegundo oscilan.

Comparando la Figura 5.7 con la Figura 5.8 se deduce que es el thread que está en la llamada MPI del primer proceso molesta al thread del proceso 1 que tiene su CPU. Este hecho provoca esa bajada de ciclos por microsegundo que supondrá que ese thread deba estar más tiempo ejecutando para hacer la misma cantidad de trabajo.

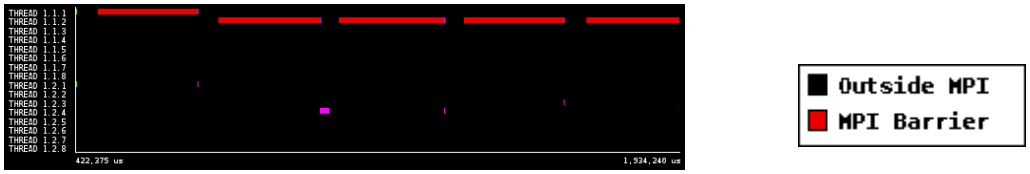

Figura 5.8: MPI calls del caso TALP+LeWI+LeWI-MPI

De esta manera queda comprobado que los valores aportados por el módulo son fidedignos en este caso. Es decir, que la diferencia entre los tiempos de cómputo del proceso P0 de las gráficas 5.2 y 5.3 es producida, por una bajada en los ciclos por segundo, como se observa en las traza 5.7.

5.3. Alya

Como se ha mencionado anteriormente, en la sección 5.1.3, Alya es una aplicación de simulación física y para este proyecto se utilizará un caso que simula la interacción de las partículas de aire con un sólido. Para ello se analizará una iteración del solver, fase en la que ocurren las comunicaciones MPI entre procesos. A su vez se cuantificará la diferencia entre las métricas producidas por Extrae y el módulo. Una vez ejecutada la aplicación junto a DLB y generadas las trazas, se observará en ellas, utilizando la configuración Custom de Paraver 5.3.1, cómo se activan las zonas de Monitoring 4.8.

También [V] se añadirán las siguientes lineas a su config.in[VI]:

```
DLB_HOME= #Apuntando al path donde este la versión compilada de DLB
CSALYA   := $(CSALYA)   -DALYA_DLB  -I$(DLB_HOME)/include
EXTRALIB   := $(EXTRALIB)  -L$(DLB_HOME)/lib  -ldlb
 ↪  -Wl,-rpath,$(DLB_HOME)/lib
```

Se utilizará la herramienta Paraver[15], proporcionada por el BSC, para mostrar las trazas con Extrae [5] y generar vistas personalizadas que ayudarán a comparar los datos. Paraver mantiene la compatibilidad con todas las trazas generadas por Extrae, así que se utilizará la última versión disponible en la web y cuyo módulo se llama PARAVER/latest.

5.3.1. Validación por trazas

Alya se ejecutará en un total de 8 nodos de Marenostrum 4 y se utilizará un caso en el que utilizando la dinámica de fluidos, se simula la turbulencia del aire alrededor de un avión. Se han elegido 8 nodos ya que de haber elegido menos, la ejecución tardaría demasiado tiempo en ejecutarse.

Normalmente, Alya se ejecuta en casos que requieren mucho poder computacional, hecho que dificulta generar una traza y en consecuencia su validación. Por ello, se ha decidido coger una de las iteraciones del ensamblaje, parte en la que se llevan a cabo los cálculos, y instrumentarla con las siguientes directivas.

```
subroutine nsi_bouope(itask)
...
zone_name = 'Zone.'
zone = dlb_monitoringregionregister(zone_name)
err =  dlb_monitoringregionstart(zone)


call cputim(time3)
call nsi_bouope(0_ip)
call cputim(time4)
err = dlb_monitoringregionstop(zone)
err = dlb_monitoringregionreport(zone)
...
end subroutine
```

[V] https://www.bsc.es/research-development/research-areas/engineering-simulations/alya-high-performance-computational

[VI] Entorno de compilación

Código 5.1: Código de Alya instrumentado

En el Código 5.2 se observa como la función *monitor_region_start* llama a la función *add_event* para que añada a Extrae el evento llamado monitor_region con el valor 1.

```
int monitoring_region_start(monitor_info_t * p ){
        if( !thread_spd->talp_enabled) return DLB_SUCCESS;

        add_event(MONITOR_REGION,1);
        struct timespec now;
        clock_gettime(CLOCK_REALTIME, &now);

        p->elapsed_time = now;

        talp_update_monitor(monitoringRegion);

        p->mpi_time = monitoringRegion->mpi_time;
        p->compute_time = monitoringRegion->compute_time;

        return DLB_SUCCESS;
}
```

Código 5.2: Evento de Extrae en el código

Cada vez que una iteración se complete, se imprimirá un resumen de esa iteración, información que se utilizará para comparar con las trazas. Cada vez que entra a una región, se añadirá un evento a Extrae que permitirá, en las trazas, poder delimitar la región a analizar.

En las trazas se utilizarán los archivos de configuración de Paraver siguientes:

- **ThreadState**: Archivo que indica qué threads tiene activos cada proceso MPI.
- **MPI Stats**: Configuración que muestra en qué momentos los distintos procesos llaman a alguna función de MPI. Habrá casos en los que haya otras políticas de DLB activas y por ello se deberá tener en cuenta el tiempo que pasen dentro de MPI.
- **Configuración Customized**: Configuración de Paraver que indicará los momentos en los que hay alguna región de monitorización activa. Esta configuración ayudará a ubicar la traza que se quiere extraer y que servirá para hacer la validación.

Este caso se ejecutará únicamente con el módulo implementado, sin activar ninguno de las políticas de DLB para simplificar la validación.

5.3.2. Resultados

Según se ha explicado en la sección 5.3.1, se ejecutará el programa con instrumentación y se compararán los tiempos obtenidos por las trazas con los que genera el módulo e imprime en la salida error estándar.

Teniendo en cuenta que cada proceso escribe las estadísticas por el output, se necesitará escoger uno de ellos para hacer las comprobaciones. Como el proceso 0 de la aplicación se dedica exclusivamente a esperar a los demás, lo se descartará y se utilizará el proceso 1, y dado que Extrae numera los procesos en las trazas a partir del 1, el proceso 1 será mostrado como proceso 2.

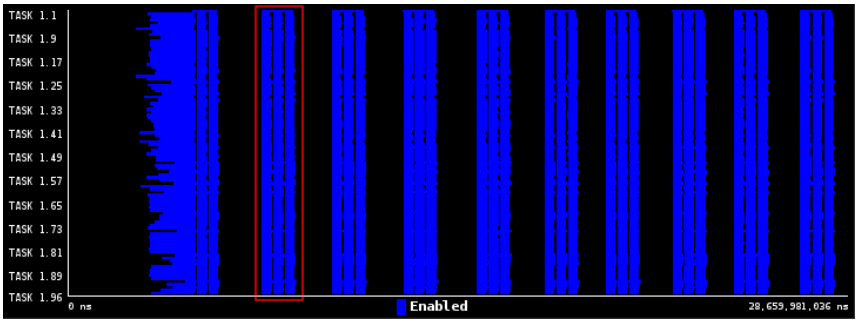

Figura 5.9: MonitoringRegions de todos los procesos

En la Figura 5.10 se muestran las zonas en las cuales la funcionalidad de MonitoringRegions está activada y como se puede observar, hay 10 iteraciones grandes y en cada una de ellas hay 3 internas. Para hacer la validación se utilizará la segunda iteración grande, dado que la primera normalmente es más irregular y dentro de ésta se analizará la primera de las tres regiones.

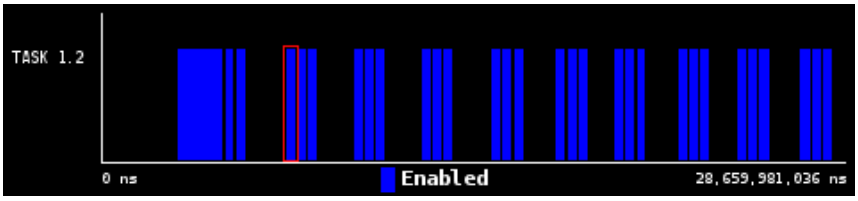

Figura 5.10: MonitoringRegions del proceso 1

En la traza 5.11 se muestra la evolución de los estados de los threads con el paso del tiempo. En ésta se observan 3 colores, el amarillo representa que un thread está en el estado ACTIVE, el rojo que está en estado IDLE y el rosa en estado CREATION. En este caso, en el que se ejecuta la aplicación sin políticas adicionales de DLB, se observa que se cumple una de las premisas en las que se basa la implementación: cuando un thread entra en una llamada MPI se contabilizará como si todos los threads entraran en ella. (Sección 4.7.1)

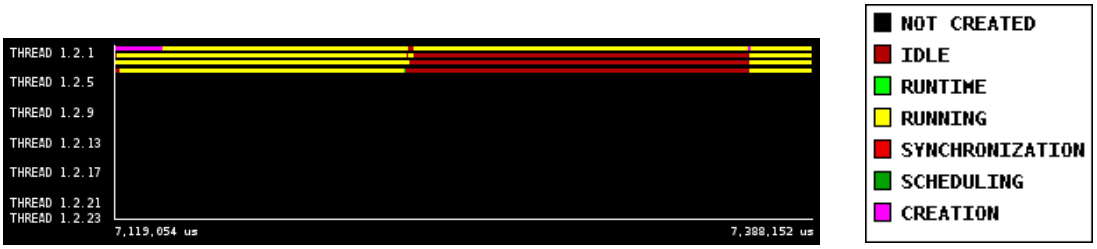

Figura 5.11: Thread State del proceso 1

En la traza 5.12 se observa, que este proceso tiene un thread en una llamada MPI

durante gran parte del tiempo. Más en concreto, es una MPI_Barrier en la que perderá bastante tiempo. Aunque esta región también consta como active en la Figura 5.11, en realidad está dentro de MPI así que lo descontaremos del modo Active.

Figura 5.12: Llamadas MPI bloqueantes del proceso 1

Se observa cómo la traza 5.13 es la diferencia entre la Figura 5.11 y la Figura 5.12 y muestra el tiempo que las CPUs están fuera de MPI. Posteriormente, y con el fin de poder mostrar el tiempo de las regiones azules, se observa la tabla generada a partir del histograma producido por la traza .

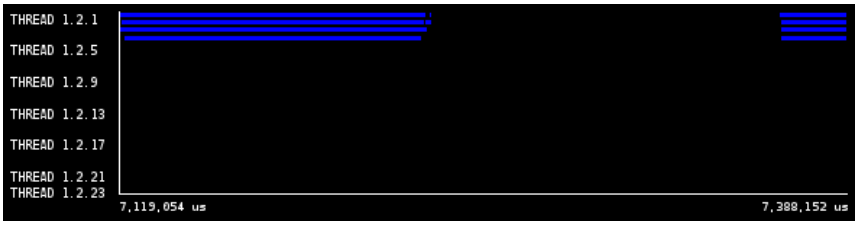

Figura 5.13: Vista restada del proceso 1

	Tiempo (s)
Total	0.5365
Media	0.1341
Máximo	0.1355
Mínimo	0.1317
Stdev	1456.81
Media/Máximo	0.99

Cuadro 5.2: Tabla del Histograma de la traza 5.13

```
1  // Iteration 1
2  DLB[NodeName:ProcessPID]: ########### Monitoring Regions Summary ##########
3  DLB[NodeName:ProcessPID]: ### Name:                       Zone
4  DLB[NodeName:ProcessPID]: ### Elapsed time :              1.663633731 seconds
5  DLB[NodeName:ProcessPID]: ### MPI time :                  6.056295884 seconds
6  DLB[NodeName:ProcessPID]: ### Compute accumulated time : 0.598243288 seconds
7  DLB[NodeName:ProcessPID]: ###      CpuSet:  [4-7]
8  DLB[NodeName:ProcessPID]: ########### Monitoring Regions Summary ##########
9  DLB[NodeName:ProcessPID]: ### Name:                       Zone
10 DLB[NodeName:ProcessPID]: ### Elapsed time :              0.267181004 seconds
11 DLB[NodeName:ProcessPID]: ### MPI time :                  0.516528272 seconds
12 DLB[NodeName:ProcessPID]: ### Compute accumulated time : 0.552205612 seconds
13 DLB[NodeName:ProcessPID]: ###      CpuSet:  [4-7]
14 DLB[NodeName:ProcessPID]: ########### Monitoring Regions Summary ##########
15 DLB[NodeName:ProcessPID]: ### Name:                       Zone
16 DLB[NodeName:ProcessPID]: ### Elapsed time :              0.267809403 seconds
17 DLB[NodeName:ProcessPID]: ### MPI time :                  0.518295620 seconds
18 DLB[NodeName:ProcessPID]: ### Compute accumulated time : 0.552925860 seconds
19 DLB[NodeName:ProcessPID]: ###      CpuSet:  [4-7]
20 // Iteration 2
21 DLB[NodeName:ProcessPID]: ########### Monitoring Regions Summary ##########
22 DLB[NodeName:ProcessPID]: ### Name:                       Zone
23 DLB[NodeName:ProcessPID]: ### Elapsed time :              0.268881966 seconds
24 DLB[NodeName:ProcessPID]: ### MPI time :                  0.517912404 seconds
25 DLB[NodeName:ProcessPID]: ### Compute accumulated time : 0.557602268 seconds
26 DLB[NodeName:ProcessPID]: ###      CpuSet:  [4-7]
27 DLB[NodeName:ProcessPID]: ########### Monitoring Regions Summary ##########
28 DLB[NodeName:ProcessPID]: ### Name:                       Zone
29 DLB[NodeName:ProcessPID]: ### Elapsed time :              0.268374134 seconds
30 DLB[NodeName:ProcessPID]: ### MPI time :                  0.516089904 seconds
31 DLB[NodeName:ProcessPID]: ### Compute accumulated time : 0.557384964 seconds
32 DLB[NodeName:ProcessPID]: ###      CpuSet:  [4-7]
33 DLB[NodeName:ProcessPID]: ########### Monitoring Regions Summary ##########
34 DLB[NodeName:ProcessPID]: ### Name:                       Zone
35 DLB[NodeName:ProcessPID]: ### Elapsed time :              0.268528675 seconds
36 DLB[NodeName:ProcessPID]: ### MPI time :                  0.516832532 seconds
37 DLB[NodeName:ProcessPID]: ### Compute accumulated time : 0.557256876 seconds
38 DLB[NodeName:ProcessPID]: ###      CpuSet:  [4-7]
```

Figura 5.14: Output Alya de TALP

En la Figura 5.2 se representa el tiempo de cómputo con la etiqueta "Total", dado que se obtiene con la suma de los tiempos de cómputo asociados a los threads del proceso. Si se compara este resultado con el obtenido en el fragmento de salida anterior, concretamente en la linea 25, se puede observar cómo los valores no difieren mucho. Es razonable que las métricas no sean exactamente iguales a las de las trazas, ya que también son diferentes los momentos en los que Extrae añade el evento y el módulo calcula las estadísticas. Concretamente en está ejecución, el módulo ha producido el siguiente error:

$$Error = \frac{|t_M - t_R|}{t_R} = \frac{|0{,}557 - 0{,}5365|}{0{,}5365} = 0{,}382 \Rightarrow 3{,}82\,\%$$

t_M= Tiempo Medido por el Módulo

t_R= Tiempo Real obtenido en las trazas

Con el fin de comparar el tiempo de MPI con el tiempo calculado por el módulo, se utiliza la tabla elaborada a partir del histograma generado por la Figura 5.12; dicha tabla es la siguiente:

	Time (s)
Total	0.1294
Average	0.1294
Maximum	0.1294
Minimum	0.1294
Stdev	0
Avg/Max	1

Cuadro 5.3: Tabla del Histograma de la Figura 5.12

Para poder comparar el tiempo de MPI del módulo con el de Extrae se deberá multiplicar el tiempo del histograma por los threads disponibles, dado Extrae solo interpreta que un thread esta en MPI y los demás están en IDLE. Dadas las mencionadas cláusulas se calculará el error producido por el módulo.

$$Error = \frac{|t_M - t_R * N_t|}{t_R} = \frac{|0{,}5179 - 0{,}1294 * 4|}{0{,}5176} = 0{,}005 \Rightarrow 0{,}5\,\%$$

t_M = Tiempo Medido por el Módulo

t_R = Tiempo obtenido en las trazas

N_t = Número de Threads

Así mismo las variaciones entre los tiempos obtenidos por las trazas y los del módulo no superan el 4 %. Con todo se puede afirmar que el módulo obtiene, para este caso, unas estadísticas precisas.

Capítulo 6

Evaluación

La evaluación de este proyecto se hará con dos fines diferentes:

- **Overhead**: comprobar, mediante una aplicación sintética, la penalización que ejerce el módulo sobre el tiempo de ejecución. Éste es uno de los test más importantes dado que el hecho de no añadir overhead es una de las características requeridas en la sección 3.1.
- **Comprobación de datos**: para comprobar la verosimilitud del output generado por el módulo en una aplicación con mucho desbalanceo. Este punto es uno de los requeridos en las premisas del proyecto especificadas en la sección 3.1.

6.1. Entorno de evaluación

6.1.1. Máquina

La evaluación del proyecto se realizará con la máquina Marenostrum4, propiedad del BSC, y que se basa en la arquitectura *x86_64*, según queda especificado en la sección 5.1.1.

6.1.2. Software

Para la evaluación se utilizarán dos entornos diferentes, cada uno relacionado con una aplicación:

- **Overhead**: se utilizará IMPI 2017.4 con GCC 7.2.0, ya que son las versiones que había disponibles la máquina cuando se inició el proyecto.
- **Evaluación de TALP**: se utilizará OpenMPI 4.0.0 con GCC 7.2.0. Se cambia de versión de MPI para comprobar si el funcionamiento es correcto también con otras implementaciones.

6.1.3. Aplicaciones

Para la evaluación del proyecto se utilizarán las siguientes aplicaciones:

- **Overhead**: se utilizará un caso sintético en el cual se puede aumentar el número de llamadas MPI por segundo.
- **BT-MZ**: benchmark[I] de la NASA que será utilizado para evaluar el módulo con un programa desbalanceado.

6.1.4. Método

Tanto para la evaluación del overhead como para la evaluación del módulo se ejecutarán las pruebas y posteriormente, en cada apartado, se interpretaran los resultados obtenidos.

6.2. Overhead

Para la evaluación del overhead del módulo, se utilizará un programa sintético que simulará un proceso llamando a MPI. Éste programa tendrá como único parámetro el número de llamadas MPI a realizar en un segundo. La aplicación tiene la siguiente estructura:

```
1  int main( int argc, char* argv[] ) {
2      double start,finish;
3      MPI_Init(NULL, NULL);
4      start=MPI_Wtime();
5
6      int world_size;
7      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
8
9      int world_rank;
10     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
11
12     struct timespec sec;
13     int times = strtol(argv[1], NULL, 10);;
14     int i = 0;
15     double aux = (double)1/times;
16     sec.tv_nsec =  aux*1000000000;
17     sec.tv_sec = times == 1;
18
19     while(i < times){
20         nanosleep(&sec, (struct timespec*) NULL);
21         MPI_Barrier( MPI_COMM_WORLD );
22         i++;
23     }
24     finish=MPI_Wtime();
25     MPI_Finalize();
26     printf("0;%.3f;\n", finish-start);
27     return 0;
28 }
```

Se utilizará la librería de **<time.h>** para simular tiempos de cómputo variables entre las llamadas de MPI. Tal como se aprecia en la linea 24, se utiliza *nanosleep* junto a una estructura *timespec* para hacer esa simulación. Entre estos "tiempos de cómputo"se añadirá la llamada *MPI_Barrier* que hará las veces de la comunicación MPI.

[I] Test para evaluar el rendimiento de una máquina

6.2.1. Resultados

La aplicación será ejecutada en Marenostrum 4, y se ejecutará con un proceso y con una CPU. Para poder comprender si existe un overhead significativo, se ejecutará el caso con 128, 256, 512, 1024, 2048 y 4096 llamadas MPI por segundo. Para cada uno de estos valores se efectuarán dos ejecuciones, una de ellas activando el módulo y DLB y la otra sin DLB ni el módulo, y para cada una de las ejecuciones se repetirá diez veces para obtener la media y su desviación, de esta manera se dispondrá de valores más precisos.

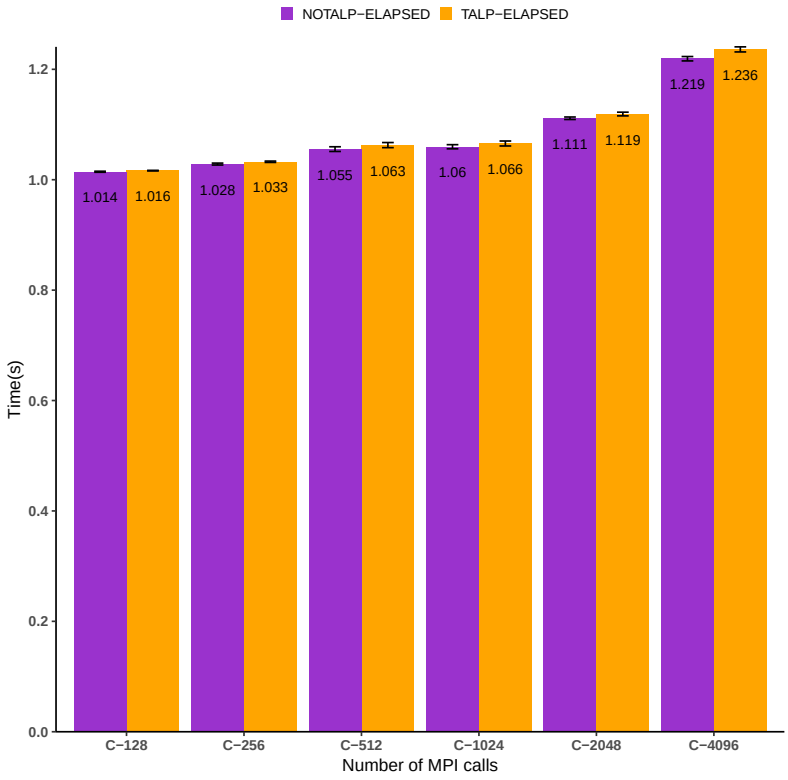

Figura 6.1: Gráfica de las ejecuciones del caso sintético

La Figura 6.1 muestra cómo se comporta la aplicación cuando se ejecuta con DLB y con el módulo; en ella se observa el overhead provocado por MPI y el módulo. La aplicación debería durar un segundo idealmente, es decir, si no tuviese overhead y se observará cómo, dependiendo del caso, tiene los siguientes overheads:

	MPI Overhead	MPI+TALP Overhead	Diferencia
C-128	1.40 %	1.60 %	0.20 %
C-256	2.80 %	3.30 %	0.50 %
C-512	5.50 %	6.30 %	0.80 %
C-1024	6.00 %	6.60 %	0.60 %
C-2048	11.10 %	11.90 %	0.80 %
C-4096	21.90 %	23.60 %	1.70 %

Cuadro 6.1: Tabla de Overheads

Según se observa en la Tabla 6.1, en la columna "Diferencia", el overhead que causa

el módulo no es alto, es decir, que gran parte del overhead causado en la aplicación tiene su origen en las comunicaciones MPI y no en el módulo. Con ello queda explicitado que se ha podido cumplir uno de los requisitos del proyecto, que es no causar un overhead significativo durante la ejecución, tal y como se había planteado en la sección 3.1.

6.3. Evaluación TALP

Para el segundo caso de pruebas se utilizará BT-M[II], un benchmark de la NASA para entornos HPC, que está escrito en Fortran 77 y tiene un desbalanceo genérico, es decir, que se ha explicitado en el benchmark. Cabe destacar que es también un benchmark CPU intensive[III] y por ello servirá para esta prueba.

Este benchmark puede ser configurado para distintos tamaños de problema, aunque en este test se utilizará el problema de tamaño B porque es uno de los tamaños estándar. Las ejecuciones serán realizadas con cuatro procesos MPI para que haya diferentes grados de desbalanceo entre ellos. Además, cada proceso tendrá dos CPUs con las que ejecutarán Nanos++; se eligen dos CPUs para utilizar el modelo de memoria compartida y se ejecuta el test con Nanos++.

Se ejecutarán tres casos de prueba para poder evaluar cómo de fidedignas son los resultados cuando el BT-MZ se ejecuta con DLB y sus políticas. Para ello se utilizarán los casos siguientes:

- **TALP**: Se ejecutará la aplicación con el módulo.
- **TALP+LeWI**: El mismo entorno que el caso anterior, añadiendo LeWI.
- **TALP+LeWI+LeWI-MPI**: El mismo entorno que el caso anterior, añadiendo LeWI-MPI.

6.3.1. Resultados

Según se observa en el la Figura 6.2, la aplicación mejora en el transcurso de los casos, ya que se reduce el tiempo de ejecución, y como era previsible, el tiempo de cómputo es similar en los tres casos. También se observa que parte del tiempo de MPI del primer caso se destina a reducir el tiempo de ejecución de los otros. Este hecho es consecuencia de la activación de las políticas LeWI y LeWI-MPI.

[II] https://www.nas.nasa.gov/publications/npb.html
[III] Benchmark que se encarga de evaluar el rendimiento de las CPUs

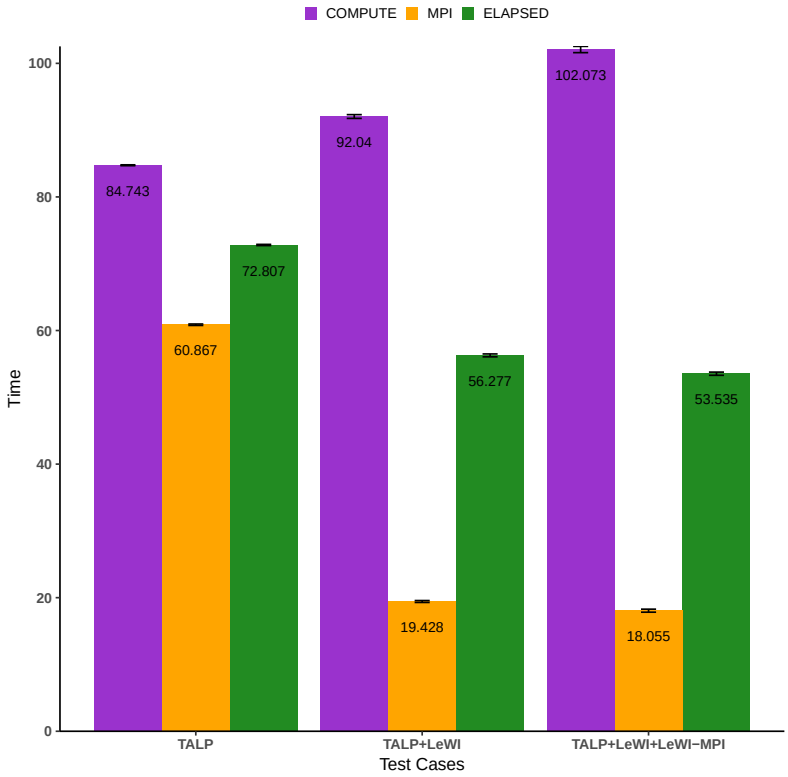

Figura 6.2: Ejecución de BT-MZ

En la Figura 6.2, el tiempo de cómputo va aumentando, hecho puede deberse a la misma razón que la expuesta en el apartado 5.2.3, es decir, por bajadas en el rendimiento de las CPUs prestadas por los otros procesos. Como se prevé, antes de ejecutar el test, el el tiempo de ejecución disminuirá con el transcurso de los casos. Este hecho reafirma que las métricas obtenidas son fidedignas como se especifica en la sección 3.1.

Capítulo 7

Planificación

En esta sección se tratará la planificación temporal del proyecto que abarcará febrero hasta junio de 2019. Además, se planificarán las horas necesarias que se dedicarán a cada etapa, contemplando las posibles desviaciones que mostrarán en forma de horas adicionales.
A pesar de esta planificación, ya cuenta con un margen de error , también pueden surgir imprevistos que sobrepasen el margen proporcionado, y que obligarían a realizar un reajuste en la planificación final.

7.1. Planificación Inicial

7.1.1. Especificación de tareas

En este apartado se especificarán cada una de las tareas que conforman el proyecto y que abarcan tanto las horas dedicadas a la gestión del proyecto como el completo desarrollo del mismo.

Gestión del proyecto (Coste en horas: 65)

La asignatura Gestión de Proyecto se divide en 4 entregables, considerados tareas, y a los que se ha asignado la siguiente carga de horas:

- **Definición del alcance y contextualización**: 20 horas
- **Planificación temporal**: 10 horas
- **Gestión económica**: 15 horas
- **Documento final y presentación**: 20 horas

En total, suman 65 horas y, aunque la asignatura Gestión de Proyectos tiene una carga de trabajo de 75 horas, esa diferencia de 10 horas permitirá disponer de una bolsa de horas por si alguna tarea necesita más tiempo de forma inesperada.
Se utilizará LaTeX para redactar las entregas, GitHub para mantener el control sobre las versiones y Gantter para hacer los diagramas de Gantt.

Puesta a punto del entorno (Coste en horas: 20)

En esta tarea, el desarrollador pondrá a punto el ordenador proporcionado por la empresa, con el objetivo de disponer del entorno necesario para iniciar el proyecto, y para conseguirlo se hace necesario realizar tareas como instalar el sistema operativo, bajar los compiladores necesarios y descargar el código actualizado de la librería. También le serán proporcionados diferentes periféricos con los que podrá trabajar con más eficiencia y se le otorgará un espacio para que pueda desarrollar el proyecto.
El desarrollador dedicará una pequeña parte del tiempo a contactar con los grupos de soporte para que le den acceso a los sistemas y máquinas. A esta tarea se le dedicarán unas 20 horas dado que deberá ser un proceso rápido.

Familiarización del modelo y DLB (Coste en horas: 60)

Inicialmente el desarrollador estudiará OmpSs, el modelo de programación paralela utilizado para el proyecto, con el fin de poder entender códigos que lo utilicen. Además, se familiarizará con la estructura de DLB con el fin de poder mantener la modularidad de la librería y entender perfectamente su funcionamiento, imprescindible si desea acoplar correctamente el modulo implementado. También contará con el personal de soporte, que en caso de tener dudas, le ayudaría a resolverás para que pueda seguir avanzando según la planificación.

Además de entender el código, también deberá ejecutar alguna prueba con alguno de los programas de ejemplo, con el fin de entender el funcionamiento de la librería y del modelo de programación, por lo que necesitará un ordenador con el que poder entender los códigos necesarios, además de los compiladores.

Diseño e implementación (Coste en horas: 205)

Esta fase, que incluye las tareas principales, busca el desarrollo del módulo de estadísticas, que deberá ser capaz de interponerse en las llamadas de MPI con la finalidad de recolectar las estadísticas en tiempo de ejecución y guardarlas para su posterior utilización. Este proceso tendrá las siguientes subtareas:

- Requisitos: esta etapa consistirá principalmente en realizar reuniones en las que se abordará la formulación del problema. Además, se especificará qué requisitos debe cumplir la propuesta de desarrollo, y que serán proporcionados por el equipo de desarrollo actual de DLB.

- Especificación: segunda etapa en la que el desarrollador irá formulando una especificación para el módulo y a verificar por el equipo de desarrollo, que será el que evalúe si realmente cumple con los requisitos establecidos en las etapas iniciales.

- Implementación: última etapa a la que se dedicará gran parte del tiempo dada la dificultad de la misma. En esta fase, el desarrollador producirá todo el código que se acoplará en la librería.

Validación (Coste en horas: 65)

En esta última etapa, el desarrollador deberá probar que el módulo recoge métricas y que se corresponden a las reportadas por las trazas, que serán confeccionadas, y poste-

riormente visualizadas gracias a Extrae y Paraver, que son herramientas proporcionadas por el BSC.

Evaluación (Coste en horas: 70)

Consiste en observar si el módulo causa overhead en la aplicación en la ejecución. Además, también se ejecutará una aplicación para comprobar si los resultados reportados por el módulo son fidedignos.

Redactado de la memoria (Coste en horas: 40)

Última etapa que se corresponde con la recogida del material necesario para la elaboración de la memoria y la posterior defensa del proyecto. Con esa finalidad se utilizará Github para mantener el control de versiones, LaTeX para el redactado de la memoria y LibreOffice para la elaboración de las diapositivas.

7.1.2. Recursos

Los recursos utilizados durante el desarrollo del proyecto, clasificados además por tipologías, son:

7.1.3. Hardware

- **1.1** - Portátil Dell Latitude 7480, 16 GB RAM, i7 7600U.
- **1.2** - Periféricos adicionales: Monitor, teclado, ratón.
- **1.3** - MareNostrum 4: Superordenador basado en arquitectura x86_64 que cuenta con 3456 nodos de cómputo. Cada uno de los cuales dispone dos procesadores Intel Xeon Platinum 8160, con 24 cores cada uno.

7.1.4. Software

- **2.1** - **Vim**: Editor de texto para implementar el código.
- **2.2** - **Extrae**: Librería que tracea los programas.
- **2.3** - **Paraver**: Visualizador de trazas de Extrae.
- **2.4** - **LibreOffice**: Editor de textos.
- **2.5** - **ShareLatex**: Editor y compilador de LaTeX.
- **2.6** - **Gantter**: Herramienta para hacer diagramas de Gantt.
- **2.7** - **Git**: Programa que permite el control de versiones.
- **2.8** - **Mozilla Thunderbird**: Programa para gestionar correos.
- **2.9** - **Atenea**: Plataforma utilizada en el curso de Gestión de Proyectos.

7.1.5. Recursos Humanos

Los recursos humanos utilizados en el proyecto especificados en el sección 1.3 son:

- **3.1 - Desarrollador**
- **3.2 - Director y Codirector**
- **3.3 - Personal de Soporte**

7.1.6. Previsión Temporal

Aquí se mostrará de manera global el volumen de horas que se dedicará a cada tarea, así como sus dependencias. También cabe destacar que esa información se especificará en el diagrama de Gantt anexado [7.1] y que la jornada laboral será de 8:00 a 14:00, es decir 6 horas diarias hasta la primera semana de Julio. Esta previsión se ha realizado para que, si surge algún imprevisto entre las tareas, se pueda reaccionar a tiempo y no sea necesario alterar las deadlines generales.

Código	*Tarea*	Horas	Dependencia	Recursos
1	*Gestión del proyecto*	65	-	1.1|1.2|2.4|2.5|2.6|2.8|2.9|3.1|3.2
2	*Puesta a punto del entorno*	20	1	1.1|1.2|2.1|2.8|3.1|3.2
3	*Familiarización del modelo y DLB*	60	2	1.1|1.2|2.1|2.7|2.8|3.1|3.3
4.1	*Diseño del módulo*	50	3	1.1|1.2|2.4|3.1|3.2
4.2	*Implementación*	90	4.1	1.1|1.2|2.4|2.7|2.8|3.1
4.3	*Validación de los resultados*	65	4.2	1.1|1.2|2.1|2.2|2.3|2.7|2.8|3.1|3.2
5	*Evaluación*	70	4.3	1.1|1.2|2.1|2.2|2.3|2.7|2.8|3.1|3.3
6	*Redactado de la memoria*	40	5	1.1|1.2|2.1|2.5|2.7|2.8|3.1|3.2
	Total	460		

Cuadro 7.1: Tabla resumen de tareas

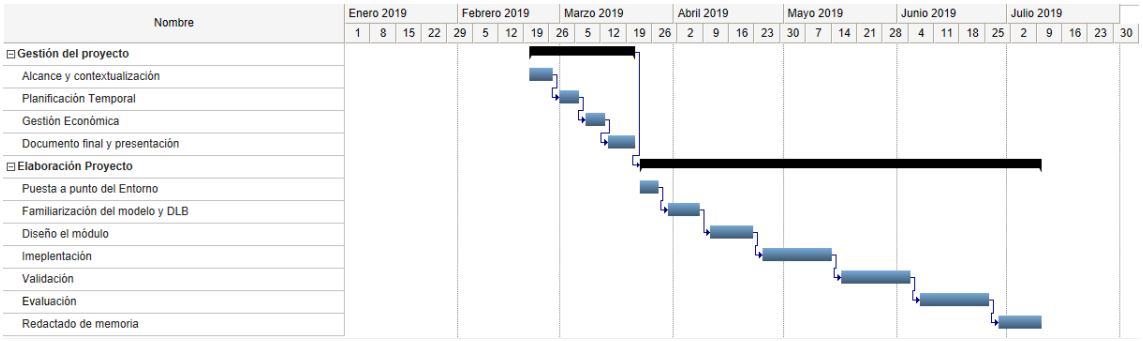

Figura 7.1: Esquema de Gant

7.1.7. Posibles desviaciones y alternativas

Es conocida la dificultad general de seguir exhaustivamente la planificación de un proyecto, ya que diferentes factores pueden alterar el desarrollo de las etapas y retrasar, en consecuencia, la finalización del mismo, y por ello es necesario establecer ciertos márgenes que eviten posibles desviaciones. Es por eso que en este proyecto se aplica la metodología SCRUM, como se especificó en el apartado 3.4, que permitirá detectar de manera rápida si alguna de las tareas no está progresando con normalidad.

La planificación propuesta tiene en cuenta estos factores y deja márgenes para la finalización de las tareas por si se produce alguna de las siguientes situaciones:

- **Desviación leve**: desviación que no alteraría el resultado del proyecto dado que la planificación cuenta con estos desvíos y hay margen suficiente para que las tareas puedan alargarse.
- **Gran desviación**: desviación pronunciada y que no se puede solucionar de manera rápida. En este caso, se deberá rehacer la planificación o, si es necesario, replantearse los objetivos para tener parte del proyecto acabado en el tiempo previsto.
- **Finalización anticipada de tareas**: Situación ideal en la que una tarea acaba antes de lo previsto, y que supondrá iniciar las tareas posteriores.

Con todo, y teniendo en cuenta que la planificación se ha hecho detalladamente, la probabilidad de grandes desviaciones es reducida o simplemente no deberían producirse.

7.2. Planificación Final

En esta sección se mostrará como ha avanzado el proyecto durante su elaboración y se presentará una tabla en la que se incluirán tanto las tareas que lo forman, ya presentadas al principio del proyecto, como su estado final. Si alguna de ellas presenta alguna desviacion o algún gasto adicional, se indicará en la siguiente tabla y posteriormente se hará una explicación asociada.

En la tabla 7.2 se muestra una relación de tareas con su estado:

Tarea	Descripción	Estado	Observaciones
1	Gestión del Proyecto	C	Ninguna
2	Puesta a punto del entorno	C	Ninguna
3	Familiarización del modelo y DLB	C	Ninguna
4.1	Diseño del módulo	C	Ninguna
4.2	Implementación	CI	Durante la implementación se dedicaron algunas horas a incorporar *major changes* de la rama master de la librería en la rama de desarrollo
4.3	Validación de los resultados	CI	Unos cambios en el runtime de OmpSs, hicieron que la instrumentación fuera imposible, y por tanto se esperó a la corrección posterior
5	Evaluación	C	Ninguna
6	Redactado de la memoria	P	Ninguna
7	Elaboración de la presentación	EC	Niguna

Cuadro 7.2: Tabla de tareas. C = Completado, CI = Completado con Imprevistos, EC = En Curso, P = Pendiente

La desviaciones constatadas son:

- **Implementación (Tarea 4.2)**: La rama *master* de la librería fué actualizada con *major changes* por lo que fue necesario dedicar horas a hacer un *merge* de ésta con la rama de desarrollo. Este *merge* ha supuesto un recargo 30 horas en la etapa de la implementación.

- **Validación de los resultados (Tarea 4.3)**: Durante el transcurso de la validación, se incorporaron en la rama *master* de OmpSs, unos cambios que inhabilitaban la instrumentación con más de un thread por proceso. Este bug se reportó al equipo de OmpSs, y que fue arreglado después de una semana de trabajo (30 horas), y esas horas fueron invertidas en hacer la memoria y adelantar así adelantar trabajo.

En referencia a la planificación inicial, se ha añadido una etapa más al proyecto, y que correspondiente a la elaboración de la presentación.

A continuación se puede observar la tabla definitiva del proyecto, en la que figuran actualizadas las horas dedicadas a cada tarea.

Código	***Tarea***	**Horas**	**Dependencia**	**Recursos**
1	*Gestión del proyecto*	65	-	1.1|1.2|2.4|2.5|2.6|2.8|2.9|3.1|3.2
2	*Puesta a punto del entorno*	20	1	1.1|1.2|2.1|2.8|3.1|3.2
3	*Familiarización del modelo y DLB*	55	2	1.1|1.2|2.1|2.7|2.8|3.1|3.3
4.1	*Diseño del módulo*	55	3	1.1|1.2|2.4|3.1|3.2
4.2	*Implementación*	120	4.1	1.1|1.2|2.4|2.7|2.8|3.1
4.3	*Validación de los resultados*	80	4.2	1.1|1.2|2.1|2.2|2.3|2.7|2.8|3.1|3.2
5	*Evaluación*	35	4.3	1.1|1.2|2.1|2.2|2.3|2.7|2.8|3.1|3.3
6	*Redactado de la memoria*	50	5	1.1|1.2|2.1|2.5|2.7|2.8|3.1|3.2
7	*Elaboración de la Presentacion*	20	6	1.1|1.2|2.1|2.5|2.7|2.8|3.1|3.2
	Total	500		

Se observa también el aumento experimentado en el numero total de horas, hecho que tendrá un impacto directo en el presupuesto, y que 9deberá ser recalculado.

Capítulo 8

Gestión Económica

En esta sección se detallan los costes del proyecto separados según los recursos. Cada tipo de recurso tendrá asociada una tabla en la que se detallarán el precio de cada recursos, así las unidades requeridas y su amortización, si corresponde.

8.1. Presupuesto Inicial

8.1.1. Costes Directos

En las siguientes tabla figurará la amortización de cada uno de los elementos que serán aportados por la empresa, y que en consecuencia deben ser amortizados. Para calcular la amortización por hora se considera que un año tiene 1800 horas laborables.

	Unidades	**Precio/Unidad (€)**	**Total (€)**	**Vida útil**	**Amortización (€/año)**	**Amortización (€/hora)**
Dell Latitude 7480	1	1500	1500	4 años	375	0.208
Pantalla Dell	1	250	250	4 años	62.5	0.035
Ratón y teclado	1	80	80	4 años	20	0.0111
Marenostrum 4	1	-	-	-	-	-
Total			1830			0.254

Cuadro 8.1: Recursos Hardware

No se contabilizarán las horas de utilización de Marenostrum 4 dado que la empresa pone la máquina a disposición del trabajador sin recargo alguno al proyecto, por lo que no se incluirá ni su amortización ni su consumo eléctrico.

	Unidades	**Precio/Unidad**	**Total (€)**	**Vida útil**	**Amortización (€/hora)**
Arch Linux	1	0	0	-	0
Libre Office	1	0	0	-	0
Nanos++	1	0	0	-	0
Vim	1	0	0	-	0
Extrae	1	0	0	-	0
Paraver	1	0	0	-	0
ShareLatex	1	0	0	-	0
Gantter	1	0	0	-	0
Git	1	0	0	-	0
Thunderbird	1	0	0	-	0
Atenea	1	0	0	-	0
Total			0		0

Cuadro 8.2: Recursos Software[I]

	Horas (h)	**Precio/Hora (€)**	**Total (€)**
Desarrollador	460	10	4600
Director	60	20	1200
Co-director	40	20	800
Personal de Soporte	30	16	480
Total			7080

Cuadro 8.3: Recursos Humanos

8.1.2. Costes Indirectos

	Meses (mes)	**Coste/Mes(€)**	**Total (€)**
Energía	6	20	120
Gastos del despacho	6	30	180
Total			300

Cuadro 8.4: Costes Indirectos

Este proyecto también tiene asociados costes que no son específicos del mismo, como el mantenimiento de las instalaciones donde se desarrollará, y que al no son publicados por el BSC, deberán ser estimados para considerarlos en cuenta en el cómputo global.

[I] Todo este software el open source y por lo tanto no genera ningún gasto.

8.1.3. Control de gestión

Dado que no es difícil sufrir desviaciones durante el desarrollo del proyecto, éstas deberán ser incluidas en el presupuesto final con el objetivo de no tener imprevistos económicos; y para detectar este tipo de anomalías se realizarán reuniones periódicas que permitirán también solucionarlas cuanto antes.

Para poder asumir estas desviaciones se presupuestarán gastos adicionales con el fin de que el presupuesto inicial no deba ser incrementado finalmente. Por tanto, se añadirá la partida de contingencia general que será un 10 % del total.

8.1.4. Tabla General

	ud	**Precio/hora (€)**	**Amortización (€/hora)**	**Horas**	**Total (€)**
Costes Directos					
Gestión del Proyecto					
Dell Latitude 7480 + Pantalla + Ratón + Teclado	1		0,254	65	16,51
Desarrollador		10		65	650,00
Director		20		10	200,00
Puesta a punto del Entorno					
Dell Latitude 7480 + Pantalla + Ratón + Teclado	1		0,254	20	5,08
Desarrollador		10		20	200,00
Personal de Soporte		16		5	80,00
Familiarizacion con el modelo y DLB					
Dell Latitude 7480 + Pantalla + Ratón + Teclado	1		0,254	60	15,24
Desarrollador		10		60	600,00
Personal de Soporte		16		8	128,00
Diseño del módulo					
Dell Latitude 7480 + Pantalla + Ratón + Teclado			0,254	50	12,70
Desarrollador		10		50	500,00
Director		20		20	400,00

Codirector		20		15	300,00
Implementación					
Dell Latitude 7480 + Pantalla + Ratón + Teclado			0,254	90	20,34
Desarrollador		10		90	900,00
Validación de los resultados					
Dell Latitude 7480 + Pantalla + Ratón + Teclado			0,254	65	16,51
Desarrollador		10		65	650,00
Director		20		20	400,00
Codirector		20		15	300,00
Evaluación					
Dell Latitude 7480 + Pantalla + Ratón + Teclado	1		0,254	70	17,78
Desarrollador		10		70	700,00
Personal de Soporte		16		17	272,00
Redactado de la memoria					
Dell Latitude 7480 + Pantalla + Ratón + Teclado	1		0,254	40	10,16
Desarrollador		10		40	400,00
Director		20		10	200,00
Codirector		20		10	200,00
Elaboración de la presentación					
Dell Latitude 7480 + Pantalla + Ratón + Teclado	1		0,254	5	1,270
Desarrollador		10		5	50,00
Costes Indirectos					300
Total acumulado					7545,59
Contingencia				10 %	754,56
Total					8300,15

Cuadro 8.5: Gastos seccionados

En la tabla anterior se puede observar, desglosado, el coste total del proyecto y se especifican, para cada una de las tareas, todos los gastos directos que tiene asociados.
A pesar de que hay tres tipos de gastos, en la tabla sólo se muestran los de hardware y los humanos dado que los de software, como se aprecia en la tabla [8.2], no tienen coste asociado.
En los gastos de hardware se observa que todas las tareas incluyen costes de ordenador y periféricos que serán amortizados por la empresa. Estas amortizaciones se han calculado por hora, dado que el desglose del proyecto está realizado especificando cuántas horas duran las tareas, y no cuántos días. Los gastos asociados a Marenostrum4, no se han añadido, al no tener un coste asociado [8.1] . En cuanto a los recursos humanos, se han estimado las horas de trabajo de cada agente para las diferentes etapas del proyecto También se ha añadido un concepto de contingencias, como se ha especificado anteriormente [8.1.3], que representará un 10 % de la suma del valor de las tareas. En la última fila de la tabla 8.5 se especifica el total, que incluye las contingencias citadas.

8.2. Presupuesto Final

En esta sección se detallará el estado del presupuesto final, las partidas que han sido modificadas y la variación respecto al presupuesto inicial.

Primeramente se especificará el cambio en las horas destinadas a recursos humanos en cada apartado del proyecto y que afectará que afectará al resto de partidas, incluida la amortización del hardware.

	Horas (h)	**Precio/Hora (€)**	**Total (€)**
Desarrollador	500 (+40)	10	5000
Director	50 (-10)	20	1000
Co-director	45 (+5)	20	900
Personal de Soporte	15 (-15)	16	240
Total			7140

Cuadro 8.6: Recursos Humanos

Como consecuencia de la actualización de horas se recalcularán los costes del proyecto, y se bajará la contingencia a un 5 % para dotar al proyecto de un ligero margen de recursos.

Recurso	**Coste**
Hardware	127,00
Software	0,00
Personal	7140,00
Costes Indirectos	250,00
Total	7517.00
Contingencia	375.85
Total Final	7892.5

Cuadro 8.7: Presupuesto Final

Nótese cómo se ha reducido en 500 euros el presupuesto inicial, disminución debida a la reducción de la partida de contingencias desde el 10 % al 5 %.

Capítulo 9

Informe de Sostenibilidad

9.1. Dimensión Ambiental

¿Has cuantificado el impacto ambiental de la realización del proyecto? ¿ Qué medidas has tomado para reducir el impacto? ¿ Has cuantificado esta reducción?
El impacto ambiental, que no ha sido cuantificado, será principalmente consecuencia del consumo eléctrico derivado del proyecto.
Para reducir ese impacto se ha procurado hacer el mínimo de pruebas posibles en el supercomputador, ya que es la fuente que genera el mayor consumo eléctrico. Además, todos los equipos que se utilicen en el proyecto, como el portátil, pueden ser reutilizados para otros fines cuando éste acabe.

¿ Si hicieras de nuevo el proyecto,¿ podrías realizarlo con menos recursos?
A partir de los conocimientos adquiridos durante el proyecto podría afirmarse que las horas utilizadas serían reducibles, así como las horas en las máquinas para hacer las pruebas.

¿ Qué recursos estimas que se usarán durante la vida útil del proyecto? ¿ Cuál será el impacto ambiental de estos recursos?
Se utilizarán clusters (Marenostrum4), periféricos y portátil, cuyo impacto ambiental se debe mayoritariamente al consumo eléctrico.

¿ El proyecto permitirá reducir el uso de otros recursos? ¿ Globalmente, el uso del proyecto mejorará o empeorará la huella ecológica?
La información que el módulo implementado puede facilitar a los usuarios durante la ejecución puede llegar a reducir el tiempo que tardan las aplicaciones en ejecutarse, hecho que, en última instancia, podría reducir el impacto ambiental.

¿ Podrían producirse escenarios que hiciesen aumentar la huella ecológica del proyecto?
No deberían producirse, dado que el proyecto no puede generar más huella ecológica, ni en su desarrollo ni posteriormente.

9.2. Dimensión Económica

¿ Has cuantificado el coste (recursos humanos y materiales) de la realización del proyecto? ¿ Qué decisiones has tomado para reducir el coste? ¿ Has cuantificado este ahorro?
Si, se ha cuantificado y ese coste está especificado en la Sección 8. Se adjunta además una tabla en la que se detallan todas las partidas y su coste asociado. Se ha intentado reducir el coste utilizando tanto material reutilizado, por ejemplo los equipos, como utilizando software libre con la intención de no provocar un sobrecoste en el proyecto, pero esa reducción no ha sido cuantificada.

¿ Se ha ajustado el coste previsto al coste final? ¿ Has justificado las diferencias (lecciones aprendidas)?
El coste final se asemeja al coste previsto, que se ha reducido en 500 euros. En este caso, como se justifica en la tabla 8.7, se habían previsto erróneamente las horas que dedicaría cada uno de los agentes implicados en el proyecto.

¿ Qué coste estimas que tendrá el proyecto durante su vida útil? ¿ Se podría reducir este coste para hacerlo más viable?
El proyecto puede tener algún coste asociado si después de la realización del mismo se descubre algún error que deba ser arreglado. Otros escenarios no generarán ningún coste asociado.

¿ Se ha tenido en cuenta el coste de los ajustes/actualizaciones/reparaciones durante la vida útil del proyecto?
No se han previsto gastos bajo esas rúbricas, pero se ha destinado una partida extraordinaria por si el proyecto necesitara un reajuste, que sería cubierto por las contingencias.

¿ Podrían producirse escenarios que perjudicasen la viabilidad del proyecto?
Como el proyecto no tiene una finalidad monetaria, no existe ningún caso en el cual la viabilidad del proyecto se vea afectada. La librería en la cual se enmarca el módulo se utiliza normalmente para fines de investigación y en ámbitos HPC en los cuales es menos frecuente buscar una ganancia monetaria.

9.3. Dimensión Social

¿ La realización de este proyecto ha implicado reflexiones significativas a nivel personal, profesional o ético de las personas que han intervenido?
A nivel profesional ha sido muy provechoso dado que muchos de los conocimientos que han sido requeridos para el mismo no habían sido adquiridos en la carrera, y por tanto han requerido una activa búsqueda.

¿ Quién se beneficiará del uso del proyecto?
La comunidad científica, más concretamente los agentes que operen en el ámbito HPC.

¿ Hay algún colectivo que puede verse perjudicado por el proyecto? ¿ En qué medida?
No, este caso no puede ser posible.

¿ En qué medida soluciona el proyecto el problema planteado inicialmente?

En el inicio el proyecto se planteaba como una ganancia en información para los desarrolladores de aplicaciones, los desarrolladores de DLB y los desarrolladores de gestores de colas. La información a conseguir, y cuya propuesta figuraba en los objetivos, se ha podido obtener, y así los agentes pueden beneficiarse del proyecto.

¿ Podrían producirse escenarios que hiciesen que el proyecto fuese perjudicial para algún segmento particular de la población?
No, este escenario no puede darse.

¿ Podría crear el proyecto algún tipo de dependencia que dejase a los usuarios en posición de debilidad?
No, este escenario es inverosímil.

Capítulo 10

Conclusiones

En este proyecto se ha constatado cómo una idea que propone una mejora de los entornos HPC, se materializa en un módulo de una librería. Además, queda reflejado que cualquiera de los niveles en la ejecución de una aplicación puede desempeñar funciones similares, tales como recoger métricas. Téngase en cuenta que, aun habiendo muchas aplicaciones que permiten recoger métricas, ninguna permite obtener esas métricas hasta finalizar la ejecución de la aplicación.

Otra de las razones impulsoras del proyecto fue el facilitar la recogida de métricas cuando se utilizan las políticas de DLB durante la ejecución las aplicaciones; además se buscó no haber de instrumentar el código para poder obtener métricas de la ejecución de las aplicaciones.

Cabe destacar que este proyecto abre el camino a posibles investigaciones futuras, con lo cual terceros agentes podrán beneficiarse y aprovechar esta implementación para otros fines de HPC. También permitiría al equipo de DLB poder interpretar esas métricas e implementar otro tipo de schedulers basados, por ejemplo, en el rendimiento de los procesos durante su ejecución, hecho que podría repercutir de manera positiva en aplicaciones con un elevado grado de desbalanceo.

Para conseguir el objetivo anterior, se ha decido poner a disposición de terceros una API con la cual se pueda acceder a toda esta información en el momento de la ejecución. Entre esos terceros agentes, los desarrolladores podrían obtener información más detallada de sus aplicaciones, y desarrolladores de gestores podrían redimensionar las máscaras, y en consecuencia resultar beneficiados de esta información.

Además, el proyecto ha sido elaborado con la intención de conseguir la máxima claridad del código y de causar el mínimo overhead posible, hecho que sido cumplido satisfactoriamente como queda demostrado en la sección 6.2. También se ha buscado respetar los principios de independencia entre módulos de DLB, hecho que ha sido clave para poder utilizar el módulo sin que los demás módulos deban estar presentes.

A nivel personal cabe destacar que este proyecto me ha permitido, en primer lugar ampliar conocimientos de ejecuciones paralelas, de los entornos HPC, y sobre todo del funcionamiento de librerías. En segundo lugar, el proyecto ha facilitado un acercamiento a

la realidad gracias al estudio de casos; y gracias a él, he podido formar parte del desarrollo de un proyecto de investigación y en consecuencia, trabajar con la presión de obtener necesariamente resultados.

Appendices

Apéndice A

Código

Se adjunta el link al repositorio donde está el código:

https://pm.bsc.es/gitlab/mrodrig3/dlb_tfg

Bibliografía

[1] E Ayguade, R Badıa y J Labarta. «OmpSs and the Nanos++ runtime». En: ().

[2] Linking Chapter 4. Compiling y Locating. https://www.oreilly.com/library/view/programming-embedded-systems/0596009836/ch04.html. [Online; accessed 15-June-2019]. 2019.

[3] Leonardo Dagum y Ramesh Menon. «OpenMP: An industry-standard API for shared-memory programming». En: *Computing in Science & Engineering* 1 (1998), págs. 46-55.

[4] Alejandro Duran y col. «OmpSs: a proposal for programming heterogeneous multi-core architectures». En: *Parallel Processing Letters* 21.02 (2011), págs. 173-193.

[5] *Extrae*. https://tools.bsc.es/extrae. [Online; accessed 20-April-2019]. 2019.

[6] Marta Garcia Gasulla. «Dynamic load balancing for hybrid applications». En: (2017).

[7] Marta Garcia, Julita Corbalan y Jesus Labarta. «LeWI: A runtime balancing algorithm for nested parallelism». En: *2009 International Conference on Parallel Processing*. IEEE. 2009, págs. 526-533.

[8] Marta Garcia, Jesus Labarta y Julita Corbalan. «Hints to improve automatic load balancing with LeWI for hybrid applications». En: *Journal of Parallel and Distributed Computing* 74.9 (2014), págs. 2781-2794.

[9] Marta Garcia Gasulla. https://upcommons.upc.edu/handle/2117/108227?locale-attribute=es. [Online; accessed 26-February-2019]. 2018.

[10] Marta Garcia Gasulla. *DLB*. https://pm.bsc.es/dlb. [Online; accessed 26-February-2019]. 2019.

[11] Github. *The world's leading software development platform Github*. https://github.com/. [Online; accessed 26-February-2019]. 2019.

[12] Jabber. https://www.jabber.org/. [Online; accessed 14-March-2019]. 2019.

[13] *MPI Tracing*. https://software.intel.com/en-us/itc-user-and-reference-guide. [Online; accessed 26-February-2019]. 2019.

[14] Wolfgang E Nagel y col. «VAMPIR: Visualization and analysis of MPI resources». En: (1996).

[15] *Paraver*. https://tools.bsc.es/paraver. [Online; accessed 20-April-2019]. 2019.

[16] Judit Planas y col. «Self-adaptive ompss tasks in heterogeneous environments». En: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. IEEE. 2013, págs. 138-149.

[17] Srinivasan Ramesh y col. «MPI performance engineering with the MPI tool interface: the integration of MVAPICH and TAU». En: *Parallel Computing* 77 (2018), págs. 19-37.

[18] Xavi Teruel. https://nanohub.org/resources/19155/download/ompss-01-quick_overview.pdf. [Online; accessed 26-February-2019]. 2019.
[19] Trello. https://trello.com/. [Online; accessed 21-February-2019]. 2019.
[20] M Mitchell Waldrop. «The chips are down for Moore's law». En: *Nature News* 530.7589 (2016), pág. 144.
[21] Wikipedia. *Scrum (software development) — Wikipedia, The Free Encyclopedia*. http://en.wikipedia.org/w/index.php?title=Scrum%20(software%20development)&oldid=884821418. [Online; accessed 26-February-2019]. 2019.
[22] Andy B Yoo, Morris A Jette y Mark Grondona. «Slurm: Simple linux utility for resource management». En: *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer. 2003, págs. 44-60.